technische universität
dortmund

fakultät für
informatik

PG 594: Big Data

– Zwischenbericht –

31. März 2016

Autoren:

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

Betreuer:

Prof. Dr. Morik, Katharina
Dr. Bockermann, Christian
Blom, Hendrik

Inhaltsverzeichnis

Teil I

Einführung

Kapitel 1

Einleitung

von Lea Schönberger

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 beschäftigt sich diese Projektgruppe mit der Verarbeitung und Analyse der Daten des durch den Fachbereich Physik auf der Insel La Palma betriebenen FACT Teleskops. 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 aktuellen Ergebnissen.

1.1 Anwendungsfall

von Michael May

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 mit Hilfe 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. Mit Hilfe solcher Kurven können dann unter anderem Supernovae klassifiziert werden [24, 88].

Das in La Palma aufgebaute First G-APD Cherenkov Telescope (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 [5].

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 Abbildung 1.1).

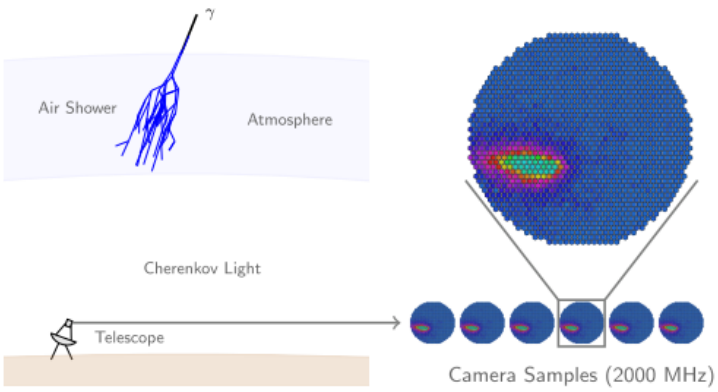


Abbildung 1.1: Visuelle Darstellung eines Gamma-Showers (oben links), welcher von Teleskopen aufgezeichnet wird (unten links) und in Grafiken der einzelnen Aufnahmen dargestellt werden kann (rechts). [22]

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 Hintergrundrauschen wird zudem erschwert, da die beiden Klassen stark ungleichmäßig verteilt sind. Bockerman et al. [22] nennen hier eine Gamma-Hadron Klassenverteilung von 1:1000 bis 1:10000. Aufgrund dieser stark ungleichmäßigen Verteilung sind 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) [47] 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 Aufbau der Arbeit

von Carolin Wiethoff

Der Zwischenbericht ist in sechs Teile gegliedert. Nach dem ersten Teil mit einleitenden Worten, Grundlagen zum Anwendungsfall und der Organisation unserer Teamarbeit folgt der zweite Teil zum 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 darauf folgenden 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.

Im dritten Teil wird der Anwendungsfall dargestellt. Neben den Analysezielen und den bisherigen Ansätzen der Physiker zur Erreichung dieser Ziele folgt eine Beschreibung der involvierten Datenformate. Es wird untersucht, wie die Daten aufgebaut sind und welche Informationen sie enthalten. Zum Schluss werden die FACT Tools vorgestellt, mit deren Hilfe die aktuelle Analysekette durchgeführt wird.

Der vierte Teil gibt einen Einblick in die Architektur unseres Endproduktes 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 bisher aus welchen Gründen am `streams`-Framework vorgenommen haben.

Der fünfte Teil widmet sich der Evaluation unserer bisherigen Ergebnisse zur Halbzeit der Projektgruppe. Außerdem geben wir einen Ausblick auf das kommende Semester und die bisher geplante Arbeit.

Das Benutzerhandbuch mit Informationen zur Installation und Ausführung im Cluster findet sich im letzten Teil.

Kapitel 2

Organisation

von Mirko Bunse

Das umzusetzende Projekt der BigData-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 erfüllen wir Abhängigkeiten zwischen den Arbeitspaketen? Wie strukturieren wir unsere Arbeit so, dass wir die 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 Abschnitt 2.1 vorstellen wollen. Warum gerade agile Verfahren für unser Projekt sinnvoll sind, wird in Unterabschnitt 2.1.1 angemerkt. Wir dokumentieren außerdem, auf welche Anwendung der Verfahren wir uns initial geeinigt haben (siehe Abschnitt 2.2) und bewerten deren Umsetzung in der PG retrospektiv (siehe Abschnitt 2.3).

2.1 Agiles Projektmanagement

von Mirko Bunse

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 Unterabschnitt 2.1.1), und wie das agile Manifest sie adressiert (siehe Unterabschnitt 2.1.2). Als kleine Auswahl agiler Verfahren stellen wir Scrum und Kanban vor (siehe Unterabschnitt 2.1.3 und Unterabschnitt 2.1.4).

2.1.1 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.

2.1.2 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 [15]:

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.

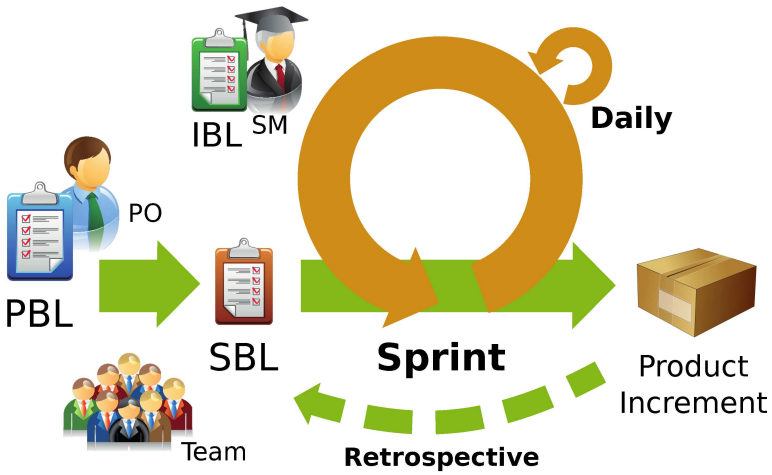


Abbildung 2.1: Der Sprint in Scrum

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.

2.1.3 Scrum

Scrum [52] 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.

Abbildung 2.1 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 managed er mit dem Product Backlog eines der Artefakte.

Der **Scrum Master (SM)** coached 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 Unterabschnitt 2.1.4).

2.1.4 Kanban

Kanban [51] 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. Abbildung 2.2 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.

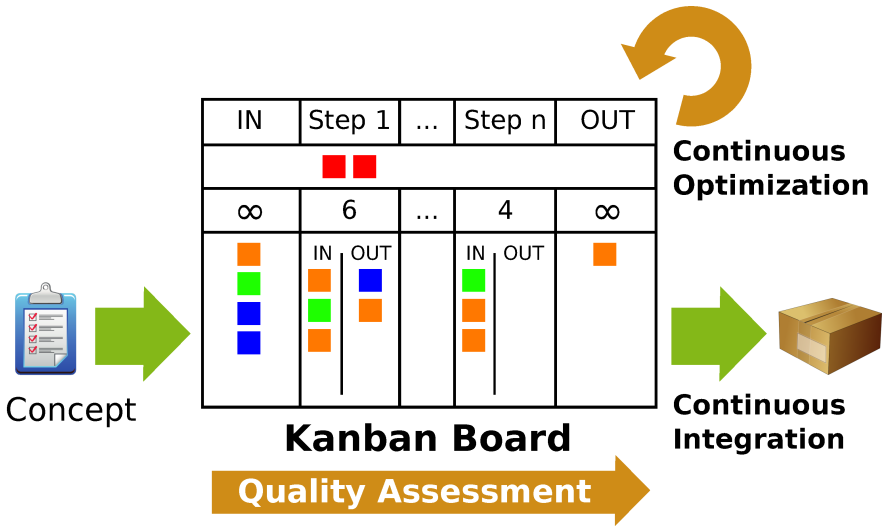


Abbildung 2.2: Das Kanban-Board

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 weiter wandern, 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 quantisiert, 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 Abbildung 2.2). 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 Unterabschnitt 2.1.3), um den Projektfortschritt zu kommunizieren. Anders als in Scrum müssen Reviews aber nicht regelmäßig abgehalten werden.

2.2 Wahl des Verfahrens

von Mirko Bunse

Scrum und Kanban (siehe Unterabschnitt 2.1.3 und Unterabschnitt 2.1.4) 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 können 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 [13]. Über die Kommentar-Funktionen dieser Projektmanagement-Software für User Stories und Tasks können wir Lösungen diskutieren und unseren Fortschritt dokumentieren.

2.3 Retrospektive der Umsetzung

von Mirko Bunse

Nach einem Semester Laufzeit der Projektgruppe können wir eine erste Bewertung unserer Umsetzung agilen Projektmanagements vornehmen. Da sich insbesondere Probleme mit der Initialisierung des Projektes erkennen lassen, wollen wir darauf gesondert in Unterabschnitt 2.3.1 eingehen. Wir wollen dazu außerdem einen Blick auf unsere Meetings werfen (siehe Unterabschnitt 2.3.2). Zusammenfassend und über diese Themen hinausgehend nehmen wir eine abschließende Bewertung vor (siehe Unterabschnitt 2.3.3).

2.3.1 Projekt-Initialisierung

Scrum fordert, dass das Team für die Vision des Endproduktes ein tiefgehendes Verständnis entwickelt. 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 initial 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.

Scrum nimmt an, dass das Team die für das Projekt nötige Expertise bereits mitbringt, im Zweifelsfall durch im Vorhinein durchgeführte Schulungen. Mit dieser Expertise kann das Projekt auch schneller initialisiert werden. Für Projektgruppen kann diese Annahme allerdings nicht vollends zutreffen, da dort große Teile dieser Expertise erst vermittelt werden sollen. Uns fehlten insbesondere Erfahrungen mit Spark und dem Streams-Framework.

2.3.2 Meetings

Das Modulhandbuch des Masterstudiengangs Informatik sieht acht Semesterwochenstunden für die Projektgruppe vor [87]. Dies ist ein wesentlich geringerer Umfang, als in einem üblichen Arbeitsleben mit acht täglichen Arbeitsstunden. Wir haben dadurch mit unserem wöchentlichen Sprint Planning Meeting einen Umfang abgedeckt, für den von Scrum ein Daily Meeting angedacht ist. Durch diesen übersichtlichen Sprint-Umfang erschien es nicht zielführend, Scrum formal durchzuführen, also ein PBL, ein SBL oder ein IBL gewissenhaft zu führen. Damit wurde aber der Großteil des Abstimmungsaufwandes in den wöchentlichen Meetings abgehandelt. Sie 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 haben wir regelmäßige Treffen der Teilteams etabliert, in denen die Arbeit erledigt und teilthemenbezogene Abstimmung erzielt wurde. Dadurch fielen die wöchentlichen Hauptmeetings sinnvoll 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. Damit haben wir noch nicht abgestimmt, wie wir unseren Entwicklungsprozess optimieren wollen.

2.3.3 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 Unterabschnitt 2.3.1). Wir sind dadurch erst recht spät aus dieser Findungsphase ausgetreten. Insbesondere waren einige Zeit lang keine sinnvollen Inkremente planbar.

Die von uns gewählte Sprintlaufzeit von einer Woche ließ eine formale Durchführung (PBL, SBL, IBL) von Scrum nicht sinnvoll erscheinen. Durch die nicht von Scrum vorgesehene Durchführung der Meetings haben wir wir viel Zeit in unseren Treffen verbraucht, wobei nicht immer alle von dieser Zeit profitieren konnten (siehe Unterabschnitt 2.3.2).

Da die Initialisierung des Projektes mittlerweile abgeschlossen ist und die in diesem Zwischenbericht vorgestellten Ergebnisse des Projektes eine gute Basis für die weitere Arbeit darstellen, haben wir jedoch eine solide Grundlage für das zweite Semester geschaffen. Wir haben ein Verständnis der Product Vision erlangt. Zukünftige Arbeitspakete werden besser planbar sein, weil wir sie im Kontext bestehender Ergebnisse betrachten können. Als Team sind wir heute eingespielter als zu Beginn der Projektgruppe.

Wir können im kommenden Semester an den hier genannten nicht-optimalen Punkten ansetzen, um unseren Entwicklungsprozess zu verbessern und damit mehr Projektziele in der uns gegebenen Zeit umzusetzen. Eine Retrospektive des bisherigen Projektmanagements stellt einen guten Startpunkt für das kommende Semester dar.

Teil II

Big Data Analytics

Kapitel 3

Einführung in Big Data Systeme

von Alexander Bainczyk

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* [65] zitiert, die in Abbildung 3.1 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.

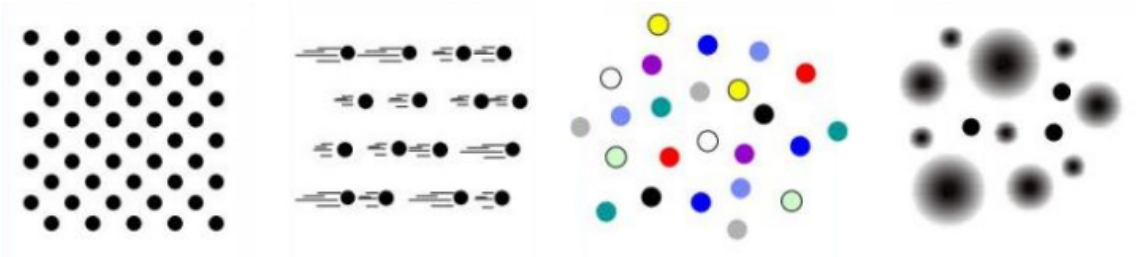


Abbildung 3.1: Veranschaulichung der ersten vier *Vs* von Big Data. Von links nach rechts: Volume, Velocity, Variety und Veracity (vgl. [25])

3.1 Nutzen von Big Data

von Alexander Bainczyk

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 Kapitel 8 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 Abschnitt 8.3 angesprochen werden und entsprechende Software, die auf die Analyse von Big Data zugeschnitten ist (s. Unterabschnitt 5.2.3).

3.2 Probleme mit herkömmlichen Ansätzen

von Alexander Bainczyk

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. Abbildung 3.2).

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. Der größte Nachteil dieses Verfahrens ist seine Kostspieligkeit, da meistens nur die Anschaffung eines neueren, leistungsstärkeren Systems möglich

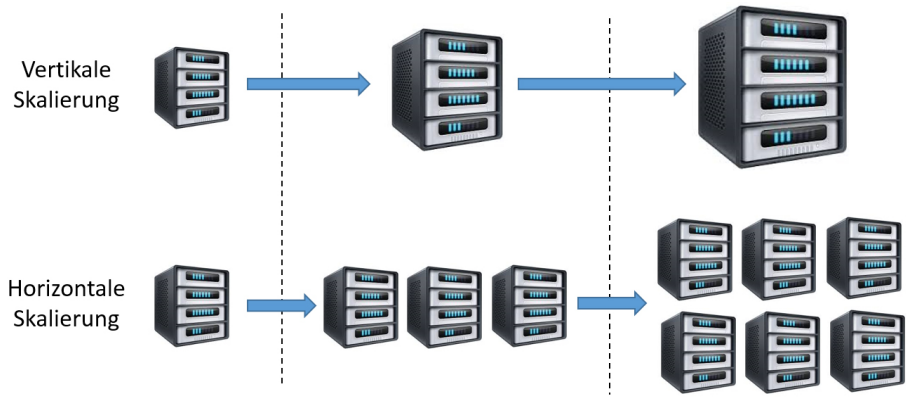


Abbildung 3.2: Arten der Skalierung

ist, wenn das alte an seine Grenzen stößt. Fürs Big Data Processing ist diese Art der Skalierung somit eher ungeeignet, da an irgendeinem Punkt es nicht mehr möglich ist, sei es aus technischer Sicht oder aus Gründen der Kosten, mehr Ressourcen in ein System einzuspeisen.

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 sind. Ein Beispiel für eine derartige, horizontal skalierbare Architektur stellt die λ-Architektur dar, die in Kapitel 4 thematisiert ist. 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 [66, Kap. 1], [91].

3.3 Anforderungen an Big Data Systeme

von Alexander Bainczyk

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 gelten, 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 mit einkalkuliert ist. Um Fehlertoleranz zu gewährleisten wird meistens auf eine Kombination aus Datenredundanz und wiederholter Ausführung von fehlgeschlagenen Teilaufgaben gesetzt. Die Feh-

lererkennung selbst geschieht zumeist auf algorithmischer Basis und soll hier nicht weiter vertieft werden [60, 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 nichtrelationale Datenbanken (s. Abschnitt 13.1, Abschnitt 13.2) 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 (Unterabschnitt 5.1.2) müssen die neuen Ressourcen entsprechend verwalten und auf Anwendungen verteilen. Eine skalierbare Architektur für Big Data Systeme wird mit der λ–Architektur in Kapitel 4 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 (Unterabschnitt 5.1.3) und Spark (Abschnitt 5.2) 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.

Kapitel 4

Lambda-Architektur

von Dennis Gaidel

Im vorangegangenen Kapitel 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. Kapitel 5) 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 wird. 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. Kapitel 13) diese Methode unterstützen, da die persistierten und indizierten Event-Daten und die zugehörige Metadaten noch keine kritische Größe erreicht hatten. Im Hinblick auf das zweite Semester und wachsenden Datenmengen (vgl. Kapitel 18) könnte die Umsetzung dieses Ansatzes vorteilhaft sein.

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 [67] eingeführt, die einen hybriden Ansatz verfolgt: Es werden sowohl Methoden zum Verarbeiten von *Batches* (also den historischen Daten, vgl. Kapitel 5), als auch zum Verarbeiten von *Streams* (Echtzeitdaten, vgl. Kapitel 6) 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. Kapitel 10), in die Datenbank überführt und indiziert (vgl. Kapitel 13).

Allgemein besteht die Lambda-Architektur (Abbildung 4.1) aus drei Komponenten: Batch Layer (Kapitel 5), Speed Layer (Kapitel 6) und Serving Layer (Kapitel 7).

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

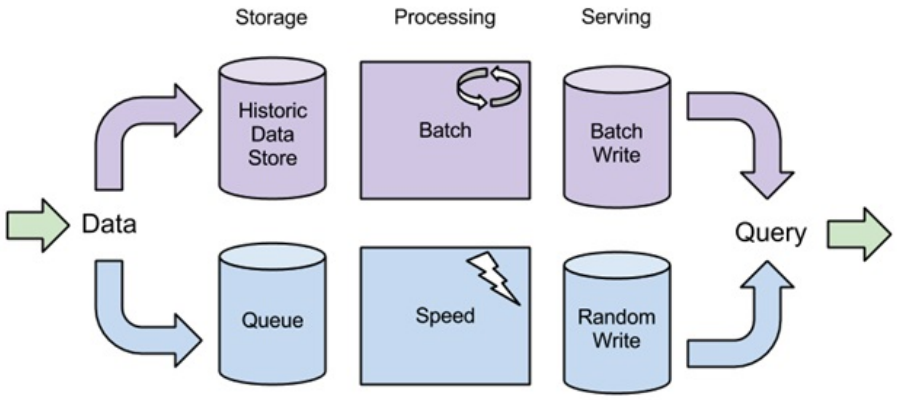


Abbildung 4.1: Lambda-Architektur [3]

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. Kapitel 11) bereitzustehen.

Der **Serving Layer** dient als Schnittstelle für Abfragen, die nach erfolgter Berechnung ein Ergebnis zur Folge haben. Diese Ergebnisse werden auf Grund 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 dem 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. Abschnitt 7.2) zur Verfügung gestellt, über die die Anwendung u.a. auch von außerhalb angesprochen werden kann.

Kapitel 5

Batch Layer

von Alexander Bainczyk

Wie im vorigen Kapitel 4 beschrieben, werden im Batch-Layer mit Hilfe 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.

5.1 Apache Hadoop

von Alexander Bainczyk

Bei dem Apache Hadoop Projekt[1] 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.

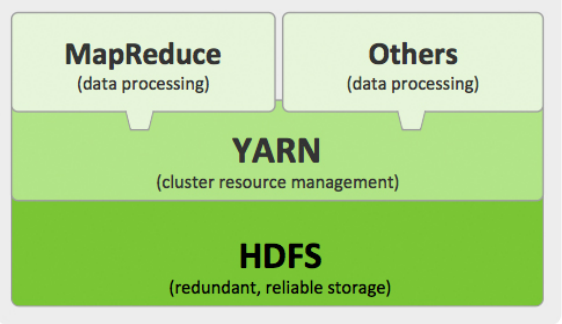


Abbildung 5.1: Architektur des Apache Hadoop Projekts. Quelle: [50]

[1] http://hadoop.apache.org/

Wie in Abbildung 5.1 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.

5.1.1 HDFS

von Alexander Bainczyk

Für den Storage-Layer in einem Rechnercluster zeichnet sich das *Hadoop Distributed File System* (HDFS) verantwortlich und basiert auf dem Google File System [42]. 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.

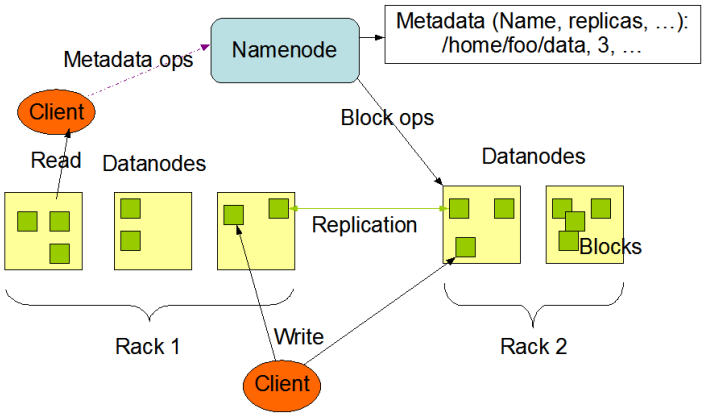


Abbildung 5.2: Funktionsweise eines HDFS Clusters. Quelle: [35]

Der Aufbau eines HDFS Clusters ist in Abbildung 5.2 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 [35].

Für die Projektgruppe wurde zu Anfang ein aus sechs Rechnern bestehendes Hadoop Cluster mit dem HDFS zur Verfügung gestellt. Das Dateisystem kommt in unserem Anwendungsfall hauptsächlich für die Persistenz der in Kapitel 10 beschriebenen Teleskopdaten zum Einsatz. Das verteilte Dateisystem erwies sich bereits als sehr zuverlässig in Bezug auf Ausfallsicherheit [42] und wird in Produktivsystemen zum Speichern und Verarbeiten mehrerer Petabyte genutzt[2], womit es eine solide Grundlage für den Anwendungszweck darstellt.

5.1.2 YARN

von Alexander Bainczyk

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].

5.1.3 MapReduce

von Alexander Bainczyk

Bei *Hadoop MapReduce* handelt es sich um eine YARN-basierte Umgebung zum parallelen

[2] http://wiki.apache.org/hadoop/PoweredBy

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 zu Nutze, 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. Kapitel 10), 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 [29].

Hadoop MapReduce hat bislang in der Projektgruppe noch 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 basiertem Framework zum verteilten, maschinellen Lernen als schwierig. Zwar existiert mit *Apache Mahout*[3] eine entsprechende, ausgereifte Lösung, nach Angaben der Entwickler wird die Entwicklung des Frameworks sich jedoch aus Gründen der Performance auf Apache Spark konzentrieren.

5.2 Apache Spark

von Dennis Gaidel

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

[3] http://mahout.apache.org/

der Berechnung auf einem einzelnen Knoten zur Folge, was insbesonders 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 100 mal schneller durchgeführt werden können [92].

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.

5.2.1 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 5.3 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, wird durch das Framework vollkommen abstrahiert und somit stark vereinfacht.

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 Transformationen zunächst einmal vom Framework sinnvoll gruppiert werden kann, um die Scans des

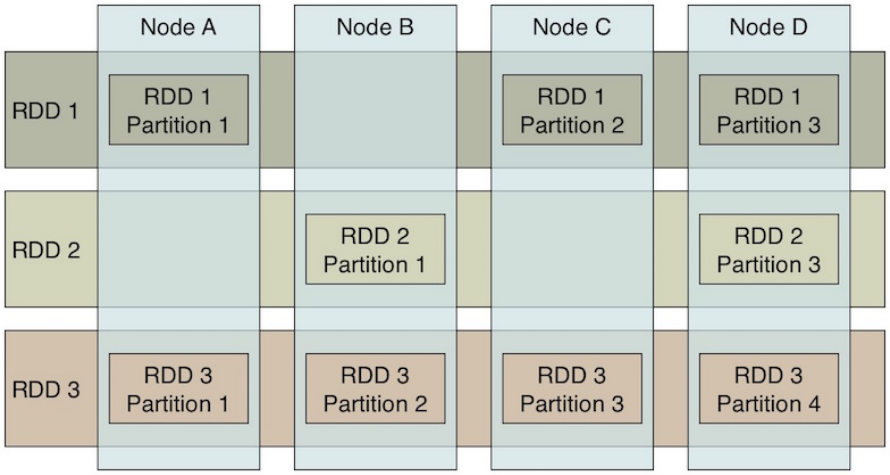


Abbildung 5.3: Verteilung der Partitionen eines RDDs auf unterschiedliche Knoten [1]

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.

5.2.2 Spark SQL

von Dennis Gaidel

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, auf Grund der höheren Abstraktion durch SQL und den zusätzlichen Typinformationen, besonders effizient zu sein.

5.2.3 Spark MLlib

von Dennis Gaidel, Carolin Wiethoff

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 die 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.

Mit Hilfe 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 5.4 dargestellt wird.

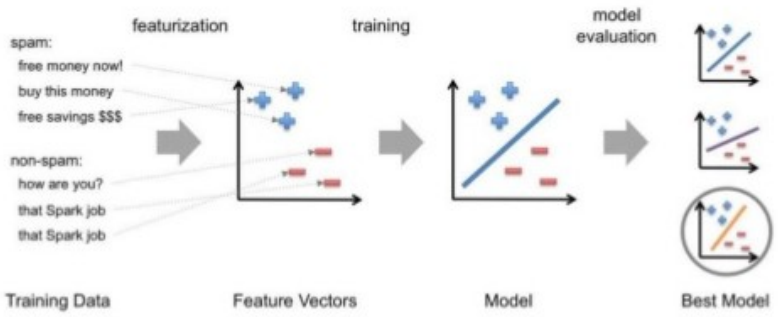


Abbildung 5.4: Maschinelles Lernen mit Spark MLlib [2]

Mit Hilfe 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.

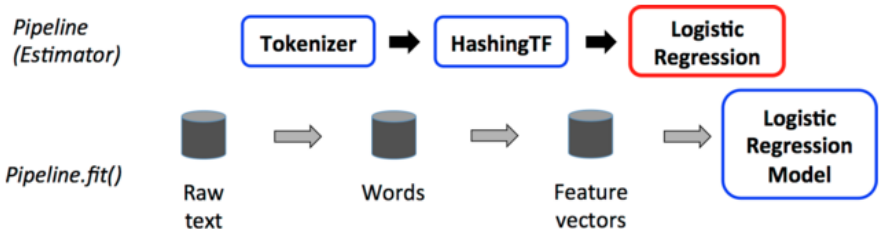


Abbildung 5.5: Pipeline-Struktur von Spark ML [9]

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. Ein Beispiel für eine solche Pipeline ist in Abbildung 5.5 zu sehen. 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.

Spark ML vs. MLlib Im Folgenden soll näher betrachtet werden, welches Paket aus Spark MLlib für unsere Projektgruppe das bessere ist. Dabei soll genauer auf die Unterschiede eingegangen werden.

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 [63] 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, welche auch im Endprodukt den Trainingsdatensatz bilden werden. Einen guten Einstieg bildet der Spark Machine Learning Library Guide [9], 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 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 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.

Kapitel 6

Speed Layer

Im Unterschied zum Batch Layer wird mit einem Speed Layer versucht die Lücke der echtzeitlichen Datenanalyse zu schließen. Neu eintreffende Daten sollen dabei direkt verarbeitet und an den Klienten weitergeleitet werden.

Im Umfang dieser Projektgruppe wurden Informationen zu gängigen Werkzeugen, die für die realzeitliche Verarbeitung von Datenströme in Frage kommen, gesammelt. Derzeit findet sich allerdings noch keine Anwendung für einige diese Tools, da zunächst die Verarbeitung von Batches im Vordergrund stand. Die Erarbeitung des Speed Layers fällt daher in die zweite Phase der Projektgruppe.

6.1 Apache Storm

von Lili Xu, Michael May

Apache Storm [10] ist ein Tool, welches zur realzeitlichen Analyse von Daten genutzt werden kann. Es ist als Open-Source Produkt verfügbar.

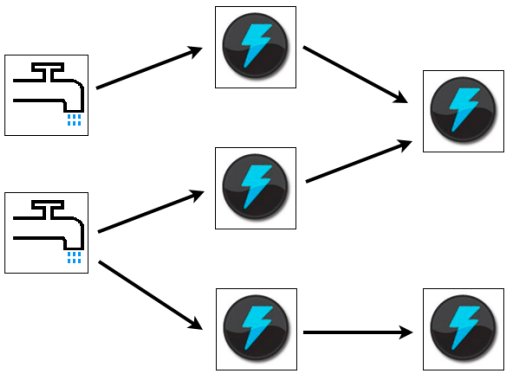


Abbildung 6.1: Beispiel einer Storm Topologie als DAG. Zu sehen sind `Spouts` (links, erste Ebene) und `Bolts` (rechts, ab zweite Ebene). Quelle: [73]

Abbildung 6.1 zeigt eine Übersicht der in Storm vorhandenen Komponenten: `Spouts` und `Bolts`. 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. Solche Aufgaben werden in Storm als Topologie bezeichnet.

6.1.1 Storm Topologien

Wie bereits erwähnt sind Topologien die Spezifikationen für Storm Aufgaben in Graphenform. Sie bestehen aus zwei Knotentypen und eine Menge von Kante, die als Datenstreams zu verstehen sind und eine endlose Sequenz von Tupeln darstellen. Abbildung 6.1 zeigt eine solche Beispiel-Topologie. In diesen Abschnitt werden die Komponenten nochmal näher betrachtet.

Spout Eine `Spout` realisiert eine Quelle für die Datenstreams und lesen im wesentlichen Eingaben ein und geben diese im Anschluss an die folgenden Knoten, in Form von Datenstreams, weiter. `Spouts` können als *reliable* oder *unreliable* markiert werden, welche das Verfahren für ein Lesefehler festlegen. Wie in Abbildung 6.1 zu sehen ist, kann eine `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.

6.1.2 Storm Cluster

Ein Storm Cluster ist ähnlich zu einem Hadoop Cluster (siehe Abschnitt 5.1), 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`. Abbildung 6.2 stellt den Aufbau eines solchen Clusters dar.

Master Node Die `Master Node` ist verantwortlich für die Verteilung des Codes, die Fehlerüberwachung und die Aufgabenverteilung. Dafür läuft im Hintergrund ein Programm namens *Nimbus*.

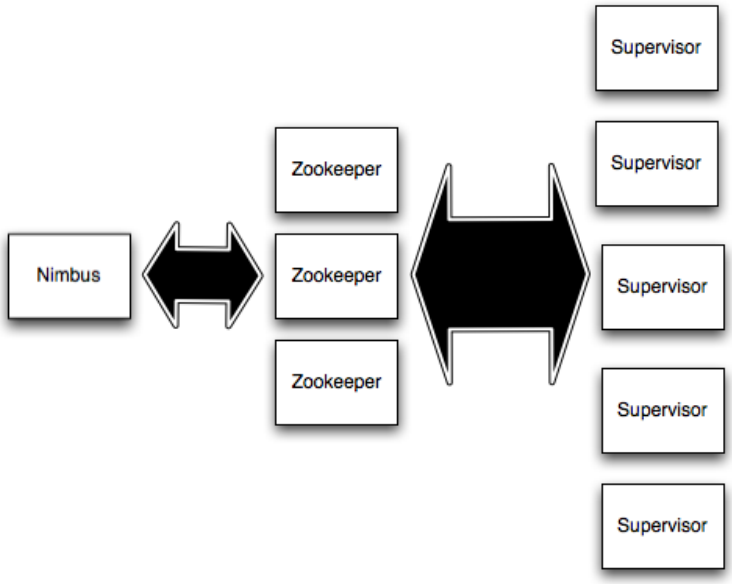


Abbildung 6.2: Aufbau eines Storm Clusters [73]

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.

6.2 Apache Trident

von Michael May, Lili Xu

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

6.2.1 Trident Topologien

Trident Topologien werden über einen Compiler in optimale Storm Topologien kompiliert. Abbildung 6.3 zeigt eine Trident Topologie, welche mit zwei Datenstreams, also bereits aus Storm bekannte `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 Informationen des oberen erhalten und mitverarbeiten kann. Zudem ist zu sehen, dass mehrere Streams über den `join` Operator miteinander kombiniert werden können.

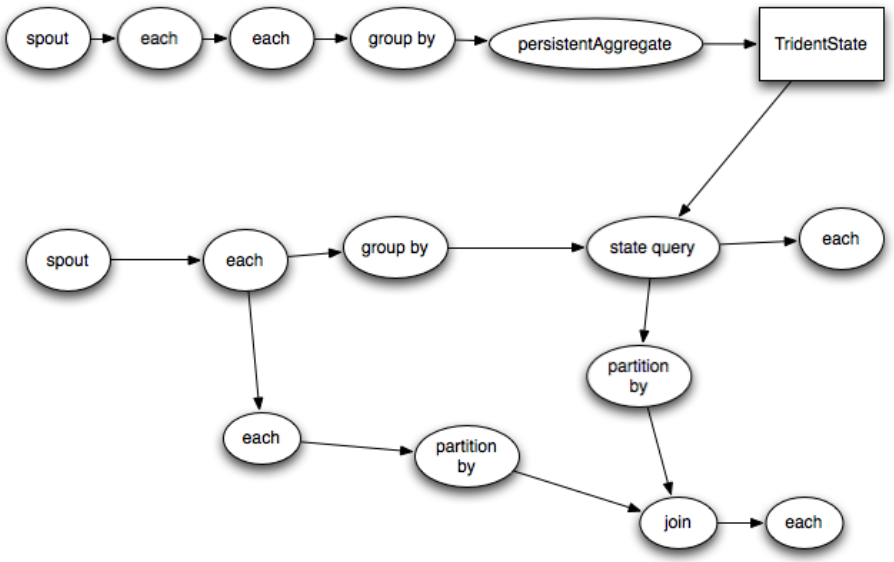


Abbildung 6.3: Beispielhafte Trident Topologie. Quelle: [74]

Abbildung 6.4 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 daher als Datenströme zu interpretieren.

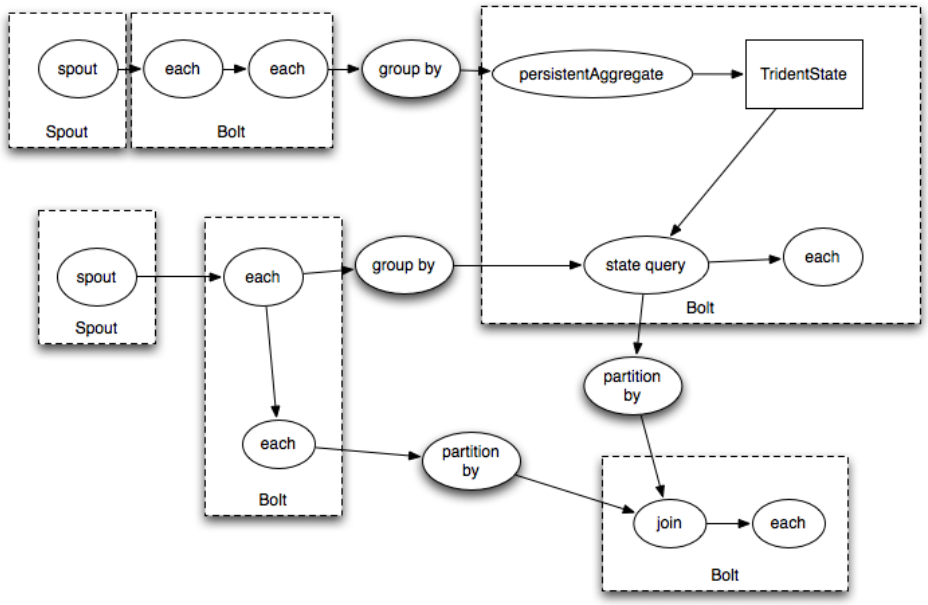


Abbildung 6.4: Abbildung 6.3 als kompilierte Storm Topologie. Quelle: [74]

6.3 Spark Streaming

von Dennis Gaidel

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 Frameworks bereit, die eine *Micro-Batch Architektur* implementiert: Streams werden als eine kontinuierliche Folge von Batchberechnungen aufgefasst, wie es in Abbildung 6.5 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 5.2 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.

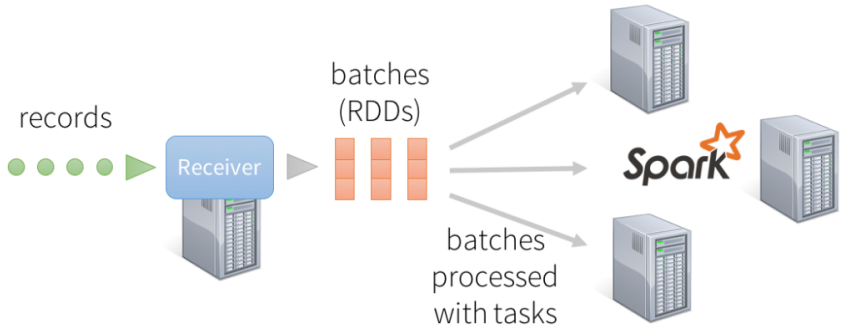


Abbildung 6.5: Verarbeitung von Datenströmen zu Batches (Quelle: https://databricks.com/blog/2015/07/30/diving-into-spark-streamings-execution-model.html)

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 6.6 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.

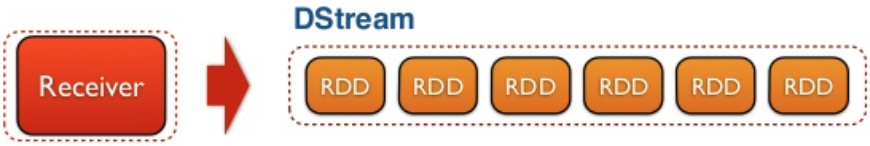


Abbildung 6.6: DStream als Datentyp zur Kapselungen von RDDs (Quelle: http://www.slideshare.net/frodriguezolivera/apache-spark-streaming)

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.

6.4 `streams`-Framework

von Michael May

Das `streams`-Framework [21, 20] 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 von den Daten eingesetzt werden können. Dafür wurde das existierende Softwarepaket Massive Online Analysis (MOA) [17] integriert und ein Plugin für RapidMiner [77] 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.

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.

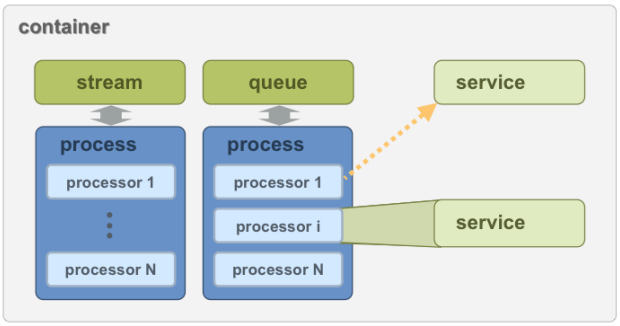


Abbildung 6.7: Schematischer Aufbau eines *Container* [20]

Abbildung 6.8: Funktionsweise eines *Stream* [20]

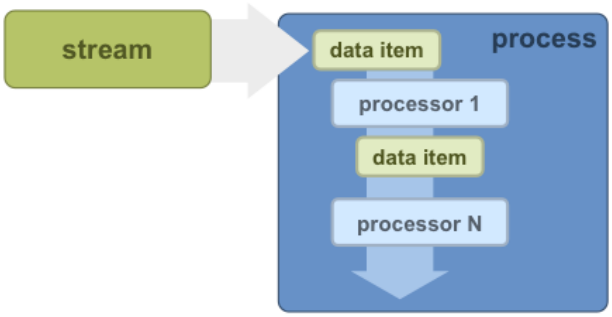


Abbildung 6.9: Arbeitsschritte eines *Process* [20]

Kapitel 7

Serving Layer

von Alexander Schieweck

Die letzte Schicht der im Kapitel 4 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 Kapitel 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 Application Programming Interface (API) erörtert.

7.1 Datenbanken

von Christian Pfeiffer

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.

7.1.1 MongoDB

von Christian Pfeiffer

Die MongoDB [70] 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 [71] 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.

7.1.2 Elasticsearch

von Lea Schönberger

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.

7.1.3 Cassandra

von Lea Schönberger

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.

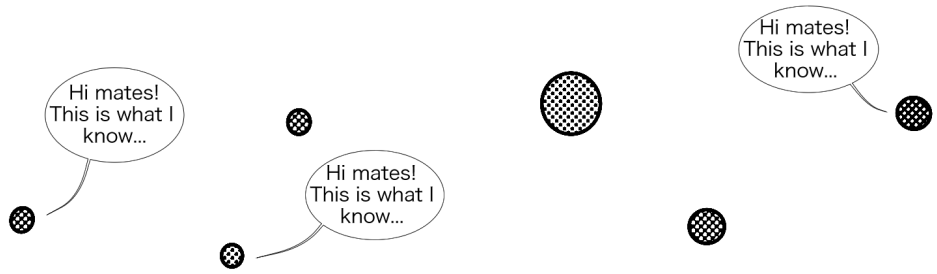


Abbildung 7.1: Veranschaulichung des Gossip Protocol. Quelle: http://blogs.atlassian.com/2013/09/do-you-know-cassandra/

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.

7.1.4 PostgreSQL

von Karl Stelzner

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 auf Grund 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 [56].

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 Operation 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 7.2 dargestellt.

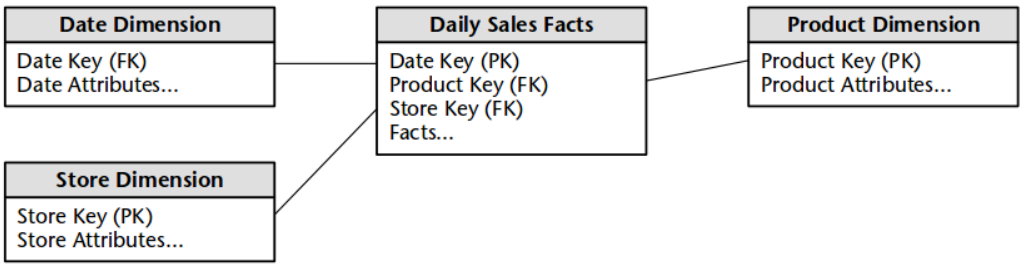


Abbildung 7.2: Ein typisches Datenbankschema nach dimensionaler Modellierung, hier am Beispiel einer Vertriebsdatenbank [56].

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

PostgreSQL

PostgreSQL [75] wird gemeinhin als das am höchsten entwickelte relationale Open-Source Datenbanksystem betrachtet [31]. 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.

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.

7.2 RESTful APIs

von Alexander Schieweck

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 von RESTful APIs vorgestellt.

7.2.1 Grundlegende Idee

von Alexander Schieweck

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 den selben 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.

Darüber hinaus 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 von einem eindeutig definierten Verhalten pro Aufruf, ohne Blick auf die Sitzungshistorie, vertrauen.

Dabei beschriebt 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 [34, 78].

7.2.2 HTTP

von Alexander Schieweck

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.

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

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 7.1: Übersicht von geläufigen HTTP Status Codes

hat. Hierbei spielt der Status Code eine besondere Rolle, da dieser eine Antwort zu Erfolg, Problemen und Misserfolg der Anfrage liefert (vgl. Tabelle 7.1).

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 [84].

7.2.3 JSON

von Alexander Schieweck

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 darüber hinaus 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 [85]. Ein Beispiel für ein solches JSON-Dokument wird in Listing 7.1 gezeigt.

```
{
        "hello": "world",
        "true": false,
        "array": [
                1, 2, 3
        ],
        "kord": {
                "x": 1.23,
                "y": 4.56
        }
}
```

Listing 7.1: Ein Beispiel für ein JSON Dokument

Kapitel 8

Maschinelles Lernen

von Carolin Wiethoff

Das letzte Kapitel im Teil *Big Data Analytics* bildet das maschinelle Lernen. Wie in Kapitel 3 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], [94] und [37] 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 3 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 Klassen 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. Abschnitt 8.2 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 Attributwerte des individuellen Datums repräsentiert, und aus einer Annotation y_i. Diese steht für die Klasse, zu der das betrachtete Datum gehört. Die Annotation 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:

$$\epsilon(h) = \mathbb{E}_{x \sim \mathcal{D}}[\mathbb{I}(h(x) \neq f(x))] \quad [94]$$

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.

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.

8.1 Ensemble Learning

von Carolin Wiethoff

Die Idee des Ensemble Learnings ist, auf viele Modelle zurückzugreifen, anstatt sich nur auf die Vorhersagen eines Modells zu verlassen. Nach Dietterich [30] 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 [62]) und *Boosting* (insbesondere AdaBoost nach [36]) im Fokus. Neben diesen beiden Quellen wurden auch Grundlagen aus [94], [30] und [76] 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.

8.1.1 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 8.1 zu sehen ist.

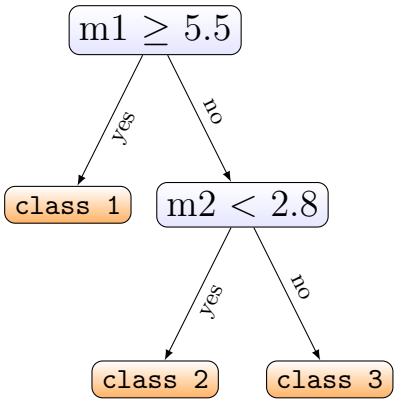


Abbildung 8.1: Beispielhafter Entscheidungsbaum

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.

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

Algorithmus 1 : Konstruktion von Zufallswäldern [62]

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 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.

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 $\epsilon_t = \Pr_{i \sim \mathcal{D}_t}[h_t(\vec{x}_i) \neq y_i]$ [1];
  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

Algorithmus 2 : AdaBoost [36]

8.1.2 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 Overfitting 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.

[1] Pr = Statistische Wahrscheinlichkeit

8.1.3 Fazit

Es gab mehrere Versuche, Bagging und Boosting miteinander zu vergleichen. Dietterich [30] 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 [76] 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.

8.2 Clustering und Subgruppenentdeckung

von Mohamed Asmi

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.

8.2.1 Clustering

Clustering [53] 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 [28]. 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 [18] 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.

8.2.2 Subgruppenentdeckung

Die bekannteste Methode zur Erkennung von Mustern mit vorgegebenen Eigenschaften ist die *Subgruppenentdeckung*. Zum ersten Mal wurde sie von Kloesgen und Wrobel [57, 58] eingeführt. Die Subgruppenentdeckung [64] 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 [23] 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 $p : P(\mathcal{A}) \times dom(\mathcal{A}) \to \{0, 1\}$, 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:

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

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* [49, 61] 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 über, wie oft war 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) [86]: 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 [79]: 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 [93]. 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, damit sie parallel bzw. verteilt arbeiten. In den folgenden Abschnitten werden diese Ansätze erläutert.

8.3 Verteiltes Lernen

von Christian Pfeiffer

Eine Grundannahme des maschinellen Lernens ist die vollständige Verfügbarkeit des Datensatzes an einem Ort. Bei großen wissenschaftlichen Datensätzen, wie sie vom FACT-

Teleskop aufgezeichnet werden, ist es aber aufgrund der schieren Größe nicht praktikabel, den Datensatz auf einem einzelnen Rechner zu halten. Dies stellt eine große Herausforderung für maschinelle Lernverfahren dar, weil Algorithmen, die Daten über das Netzwerk statt von der eigenen Platte laden müssen, potenziell deutlich langsamer sind.

Für das Problem des maschinellen Lernens auf verteilten Datensätzen gibt es bereits einige Verfahren, die in bestimmten Situationen weiterhelfen können. Die Suche nach verallgemeinerbaren Lösungsansätzen ist immer noch aktives Forschungsthema.

Im Folgenden werden zwei Verfahren vorgestellt, die das Problem von sehr unterschiedlichen Perspektiven angehen. Dies sind der Peer-to-Peer-K-Means von Bandyopadhyay et al. [14] sowie die Modellkompression für Entscheidungsbäume von Kargupta und Park [55].

Der Peer-to-Peer-K-Means. Der erste Lösungsansatz liegt in dem Entwurf neuer maschineller Lernverfahren, die direkt berücksichtigen, dass die Daten sich an unterschiedlichen Orten befinden. Der P2P-K-Means erweitert das bekannte Prinzip des K-Means (siehe Unterabschnitt 8.2.1) insofern, dass jeder Rechenknoten im Netzwerk den K-Means-Algorithmus auf seine lokalen Daten anwendet und nach jeder Iteration seine errechneten Zentren an die Nachbarn im Netzwerk versendet. Diese versuchen dann, einen Mittelwert über die lokalen und die empfangenen Zentren zu bilden, und nutzen diese Werte als lokale Zentren für die nächste Iteration. Dies wird so lange fortgesetzt, bis sich die Zentren bis zu einem festgelegten Abstand angenähert haben.

Allerdings zeigt sich, dass der Entwurf eines verteilten Algorithmus komplexer ist als der eines lokalen Algorithmus. Die Spezifikation muss folgende Aspekte auf jeden Fall umfassen:

- 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?

Innerhalb dieser Projektgruppe werden keine eigenen, verteilten maschinellen Lernverfahren entworfen, sondern die Algorithmen aus der Spark ML Bibliothek erprobt. Eine Einführung in Spark ML findet sich in Unterabschnitt 5.2.3.

Die Kompression von Entscheidungsbäumen. Eine andere Strategie besteht darin, keine verteilten Lernverfahren auf die verteilten Daten anzuwenden, sondern an jedem Rechenknoten ein traditionelles, lokales Lernverfahren einzusetzen. Dadurch wird an jedem Rechenknoten ein eigenes Modell trainiert. Diese Menge von Modellen kann dann wie beim Ensemble Learning in Abschnitt 8.1 genutzt werden. Der dort vorgestellte Mehrheitsentscheid kann so implementiert werden, dass jeder Rechenknoten die unklassifizierten Daten empfängt, sein lokales Modell darauf anwendet und die resultierende Klassifizierung an einen Koordinator schickt. Somit ist es für die reine Klassifizierung nicht zwingend notwendig, die lokalen Modelle im Netzwerk zu versenden.

Allerdings gibt es einige praktische Gründe, die das Versenden von Modellen irgendwann notwendig machen. Dazu zählt beispielsweise das Klassifizieren von Datenströmen mit hohem Datendurchsatz. Hier würde das Versenden der unklassifizierten Daten an die Rechenknoten und das Warten auf die Antwort zu hohen Latenzen führen, die einem hohen Datendurchsatz entgegenstehen.

Das Kompressionsverfahren von Kargupta und Park 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.

8.4 Statisches und Inkrementelles Lernen

von Alexander Schieweck

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.

8.5 Concept Drift und Concept Shift

von Alexander Schieweck

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.

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 beschreiben und angesprochen, wie man das Verschieben erkennen kann [32].

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. Abbildung 8.2 links und rechts). Verschieben sich die Daten jedoch so, dass ein zur Klassifikation nötiges Feature betroffen ist und die

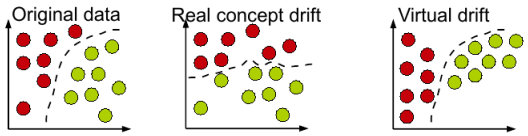


Abbildung 8.2: Unterscheidung Realer Drift vs. Virtueller Drift [39]

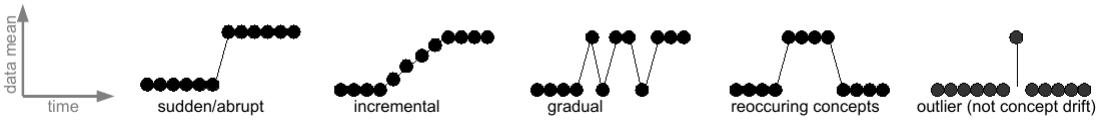


Abbildung 8.3: Schematische Darstellung vom unterschiedlichen Auftreten von Concept Drift [39]

Daten die bisherigen Unterteilungskriterien nicht mehr erfüllen, spricht man von *realem Drift* (vgl. Abbildung 8.2 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 8.3):

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.

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 Al-

gorithmen. In beiden Fällen können maschinelle Lernmethoden angewendet werden, um einen möglichen Shift zu erkennen und um die Nutzer entsprechend zu informieren [39].

8.6 Learning with Imbalanced Classes

von Karl Stelzner

Bei vielen realen Klassifikationsproblemen geht es darum, seltene Ereignisse in einer Masse aus uninteressanten Vorkommnissen zu entdecken [38]. 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 Abschnitt 1.1)

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.

8.6.1 Einfluss auf Klassifikatoren

Der Einfluss, den das Klassenungleichgewicht auf die Leistung von Klassifikatoren hat, wurde in verschiedenen Studien empirisch untersucht [54]. 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.

8.6.2 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 na-

türlicher Ansatz für die Darstellung der Performanz eines binären Klassifikators ist eine Wahrheitsmatrix, wie in Abbildung 8.4 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 verschiedene 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 8.4: Schematischer Aufbau einer Wahrheitsmatrix

Eine 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 [33] wie in Abbildung 8.5.

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

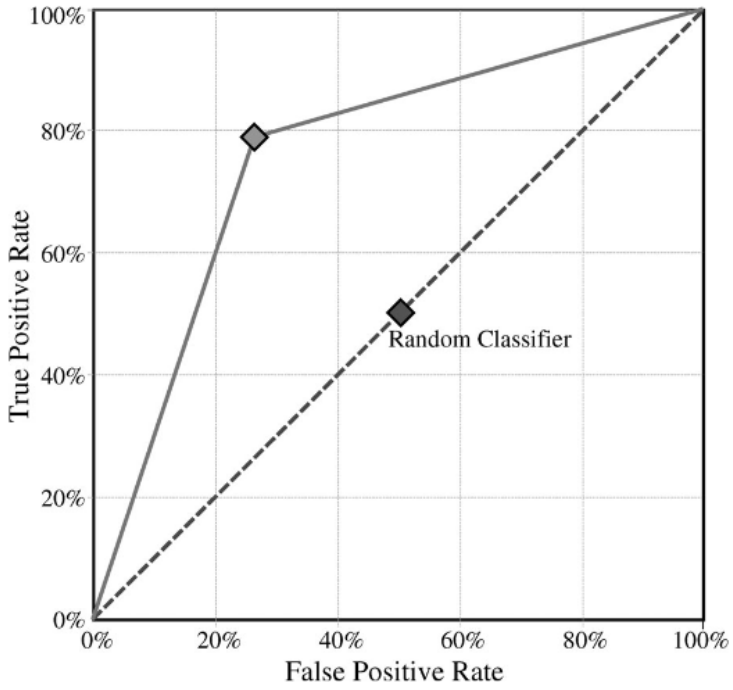


Abbildung 8.5: Eine ROC Kurve [38]

auf der anderen. In diesem Fall bietet es sich an, eine asymmetrische Metrik zu verwenden, etwa den Index of balanced accuracy (IBA) [40].

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

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.

8.6.3 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 [38, 41].

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 Abschnitt 8.1 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 (Algorithmus 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.

8.7 Feature Selection

von Mirko Bunse

Unter einem Merkmal (engl. *Feature*) versteht man im maschinellen Lernen 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.

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 Unterabschnitt 8.7.1 vorgestellt. Es wird eine formale Problemstellung aus den Eigenschaften abgeleitet, die ein „geeignetes“ Merkmal erfüllen sollte (siehe Unterabschnitt 8.7.2). Eine Übersicht der Ansätze zur Feature Selection wird vorgestellt und Qualitätsmerkmale von

Auswahl-Algorithmen werden identifiziert (siehe Unterabschnitt 8.7.3). Als prominentes Beispiel wird der korrelationsbasierte Algorithmus CFS (Correlation-based Feature Selection) nach Hall [46] intensiv betrachtet (Unterabschnitt 8.7.5). Dessen Erweiterung zu Fast-Ensembles wird ebenfalls vorgestellt (siehe Unterabschnitt 8.7.6).

8.7.1 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 Merkmale 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.

8.7.2 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 [59] prägten die Vorstellung einer optimalen Merkmalsmenge wie folgt:

Definition 8.1 (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) Wahrscheinlichkeits-Verteilung 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, das 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.

8.7.3 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.

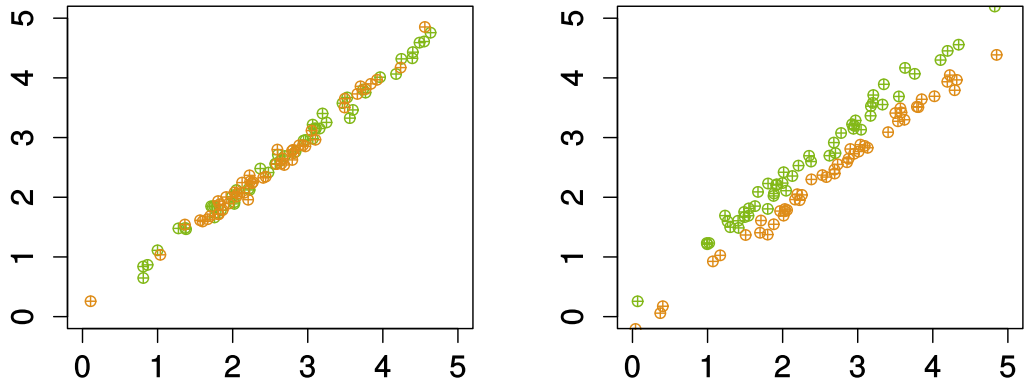
Über diese Arten hinaus existieren hybride Verfahren, die etwa Filter für eine Vorauswahl verwenden, um im Anschluss einen Wrapper die Endauswahl treffen zu lassen. Wir wollen hier Filter fokussieren, da sie das allgemein effektivste Verfahren darstellen. Durch Berücksichtigung von zusammenspielenden Features können sie bereits sehr gute Ergebnisse liefern. Die Qualität eines Algorithmus lässt sich überdies an folgenden Eigenschaften messen [81]:

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 ver-



(a) Perfekte Korrelation **(b)** Hohe Korrelation

Abbildung 8.6: Korrelation als Heuristik

wendeten 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.

8.7.4 Korrelation als Heuristik

Bevor wir in Unterabschnitt 8.7.5 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

Abbildung 8.6 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 (Abbildung 8.6a) 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 Abbildung 8.6b 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 [45].

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 [44]).

8.7.5 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 [46].

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:

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

Für das Maß *Cor* existieren verschiedene Definitionen basierend darauf, ob die Eingabe-Merkmale numerisch oder nominal sind (siehe [44]). Diese sollen hier aber nicht weiter betrachtet werden.

Beispiel-Ablauf

Abbildung 8.7 zeigt einen Beispiel-Ablauf des CFS-Algorithmus: Im ersten Schritt wird für jedes Merkmal dessen Korrelation mit der Zielvariablen bestimmt. Das Merkmal mit der höchsten Korrelation (hier `X2`) wird gewählt. In den weiteren Schritten müssen zusätzlich die Korrelationen mit zuvor gewählten Merkmalen berechnet werden, um die Redundanz

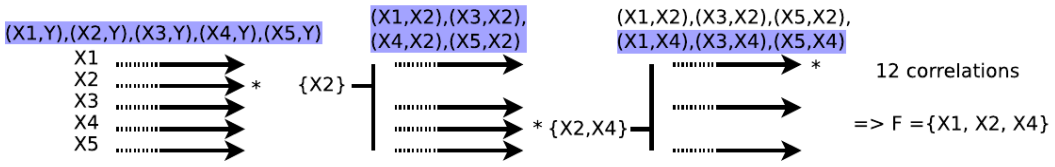


Abbildung 8.7: Beispiel-Ausführung CFS [81]

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 [46]. 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.

8.7.6 Fast-Ensembles

Um die Stabilität eines Klassifikators zu erhöhen, lassen sich mehrere Klassifikatoren zu einem Ensemble zusammenfassen (siehe Abschnitt 8.1). Die selbe Idee lässt sich auf Feature Selection übertragen, um die Stabilität der ausgewählten Merkmalsmengen zu erhöhen [80]. 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 [81] 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 Unterabschnitt 8.7.5 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 $(x_i,y_i), 1 \le i \le n, x_i \in X, y_i \in Y$ 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 $s_j(X,Y), 1 \le j \le e$ 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. Abbildung 8.8 fasst die Schätzung der Korrelation im Ensemble zusammen.

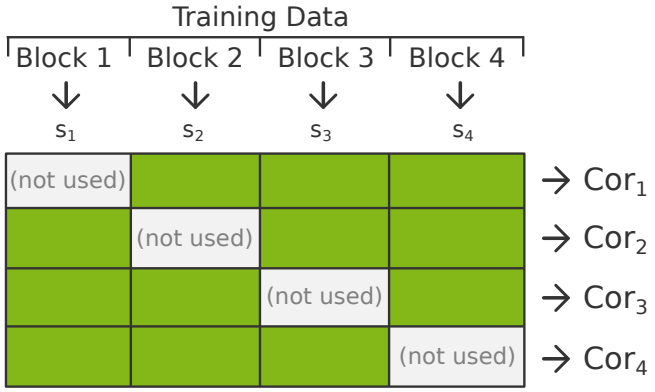


Abbildung 8.8: Berechnung von Ensemble-Korrelationen in Fast-Ensembles

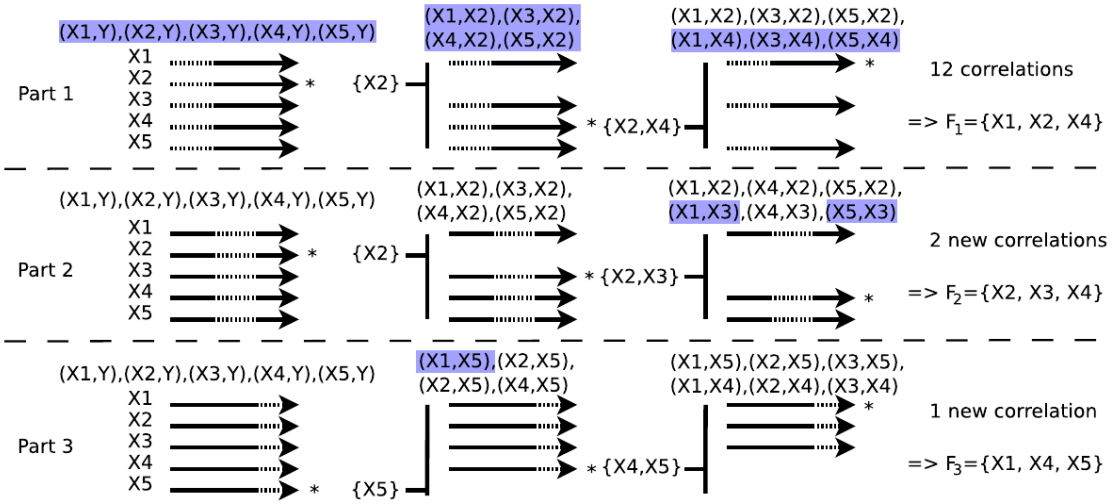


Abbildung 8.9: Beispiel-Ausführung Fast-Ensembles [81]

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.

Abbildung 8.9 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 Unterabschnitt 8.7.5 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 Unterabschnitt 8.7.3 vorgestellten Qualitätskriterien.

8.8 Sampling und Active Learning

von David Sturm

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.

8.8.1 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.

8.8.2 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.

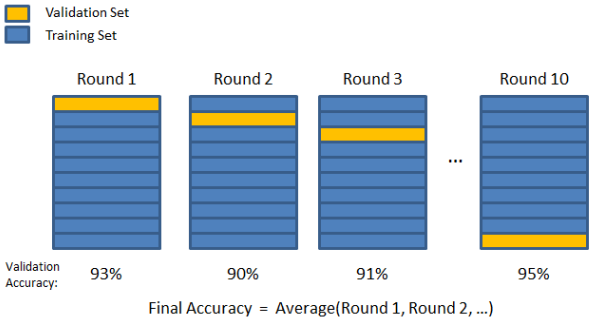


Abbildung 8.10: k-fache Kreuzvalidierung, Quelle: [26]

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 $h(x) = h(h_1(x), .., h_k(x))$

Algorithmus 3 : VLDS-Ada^2Boost [48]

8.8.3 VLDS-Ada^2Boost

Als nächstes betrachten wir nun den VLDS(Very Large Data Set)-Ada^2Boost Algorithmus. Dieser ist eine Variation des *AdaBoost*-Algorithmus 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^2Boost 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 [48]. Hierbei ist zu beachten, dass der in dem Paper behandelte Algorithmus, der Ada^2Boost 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*^{2}*Boost*-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 eben diese. 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*^{2}*Boost* 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.

8.8.4 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

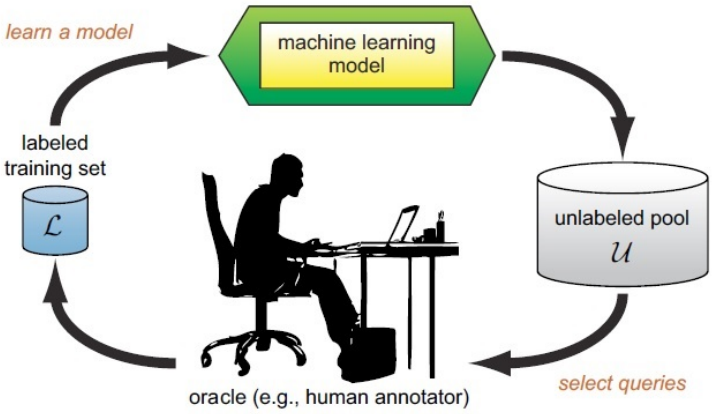


Abbildung 8.11: Active learning als Kreislauf, Quelle: [82]

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 III

Anwendungsfall

Kapitel 9

Analyseziele

von Carolin Wiethoff

Um unser Endprodukt perfekt auf die Anforderungen der Physiker abzustimmen, war es unumgänglich, sich mit den eigentlichen Analysezielen auseinanderzusetzen. In einem Treffen mit einem Repräsentanten der Physiker konnten wir mehr über den Anwendungsfall (siehe Abschnitt 1.1) erfahren und unsere Fragen dazu stellen. Im Nachhinein fassten wir das gewonnene Wissen in *User Stories* zusammen, welche nicht nur einen Überblick über die Analyseziele geben, sondern auch das Entwickeln von *Sprints* vorbereiten sollten, so wie sie in Kapitel 2.1.3 über das Projektmanagement mit SCRUM beschrieben wurden. Im Folgenden werden die aus unserer Sicht wichtigsten Analyseziele zusammengefasst, welche wir mit unserem Endprodukt ermöglichen wollen.

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 Abschnitt 7.2, für unsere Umsetzung siehe Kapitel 14) 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 10.4 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 im Endprodukt Methoden enthalten sein, mit denen man verschiedenen 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 Kapitel 8 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 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 des Endproduktes. 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 Kapitel 6.

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 unser Endprodukt 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.

9.1 Gamma/Hadron-Klassifizierung

von Michael May

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. [19] und Sharma et al. [83] 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 $Q = \frac{\varepsilon_\gamma}{\sqrt{\varepsilon_P}}$, welcher vor allem einen historischen Wert besitzt. Hierbei beschreibt ε_γ die korrekt klassifizierten Gamma-Events und ε_P die als Gamma klassifizierten Hadron-Events. Es ist vergleichbar mit der statistischen Signifikanz.

9.2 Energie-Abschätzung

von Michael May

Ein weiteres Anwendungsgebiet für maschinelles Lernen ist die Abschätzung der Energie von klassifizierten Gamma-Events. Da mithilfe der Energie viele physikalische Eigenschaf-

ten 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. [16] besagen, dass bereits das Feature `size` für eine gute Einschätzung mit Hilfe eines *Random Forest* genügt.

Kapitel 10

Datenbeschreibung

von Alexander Schieweck

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.

10.1 FITS-Dateiformat

von Alexander Schieweck

Das Flexible Image Transport System (FITS)-Format [43] 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*.

10.2 Rohdaten

von Alexander Schieweck

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 zusammen gefasst, 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!

10.3 Monte-Carlo-Daten

von Christian Pfeiffer

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.

10.4 Drs-Daten

von Alexander Bainczyk

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 Abbildung 10.1 (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 Zeitständen 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. Abbildung 10.1 (rechts)).

[6], [4], [27]

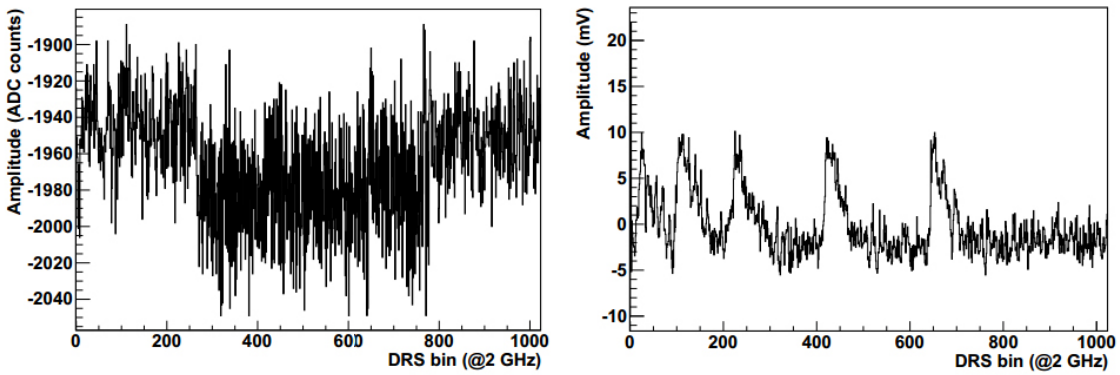


Abbildung 10.1: Event vor (links) und nach (rechts) der DRS Kalibrierung. Die Spitzen entsprechen den Signalen einer einzelnen Fotodiode. Quelle: [6]

10.5 Aux-Daten

von Alexander Bainczyk

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

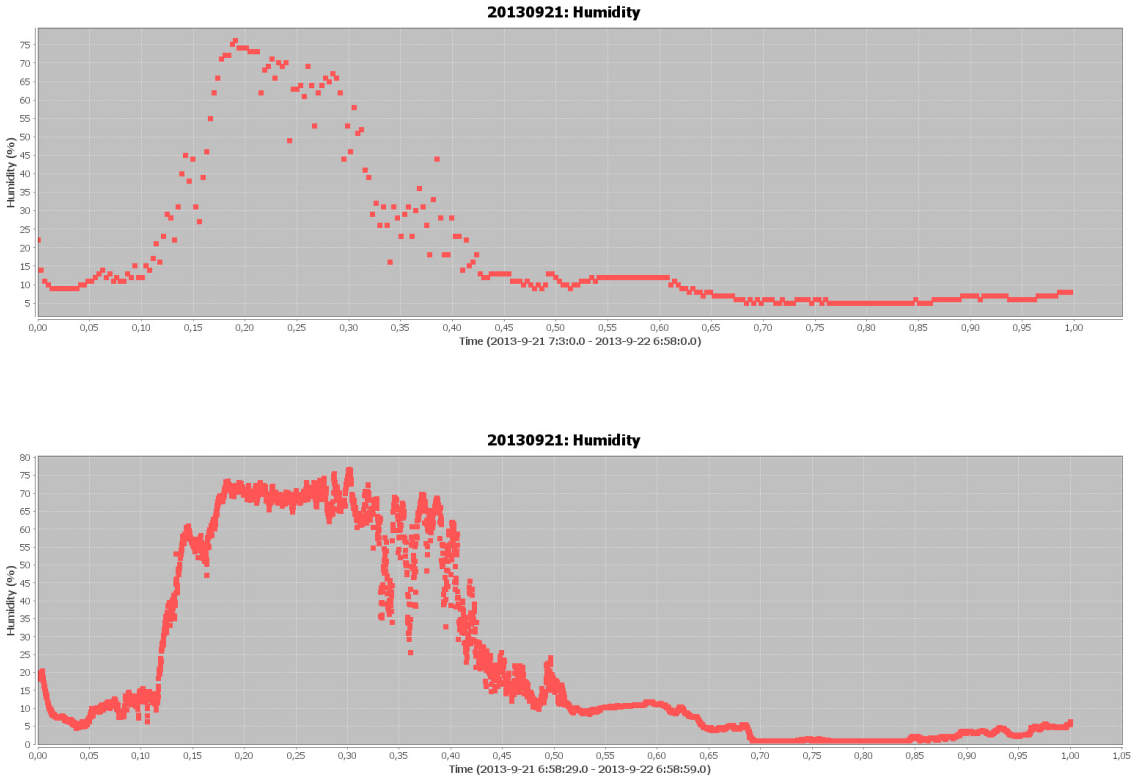


Abbildung 10.2: Statistik zur Luftfeuchtigkeit in der Nacht des 21.09.2013 aufgenommen von zwei Sensoren: TNG (oben) und MAGIC (unten)

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. 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 Abbildung 10.2 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. Abbildung 10.2).

Kapitel 11

Analyse mit den FACT Tools

von David Sturm

Für die Verarbeitung von FITS-Dateien (siehe Kapitel 10), die mit Hilfe des FACT-Teleskops aufgenommen werden, wurden die FACT-Tools implementiert. Das ist eine Erweiterung des `streams`-Frameworks.

Bei den FACT-Tools [22] wurden Inputs und Funktionalitäten für `streams` implementiert, die für die Verarbeitung der 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. Darüberhinaus ermöglichen die FACT-Tools, eine Datenanalyse mit allen Schritten, die in diesem Abschnitt erläutert werden, durchzuführen. Dazu gehören alle Vorverarbeitungsschritte sowie das Einbinden von Bibliotheken für maschinelles Lernen.

11.1 Analysekette

von Mohamed Asmi

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

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

11.1.1 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.

11.1.2 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.

Als Nächstes wird die Datenanalyse durchgeführt. Da nicht alle Attribute gleich wichtig sind, wird zuerst eine *Feature-Extraktion* durchgeführt. Dabei wird ermittelt, welche haben die Attributen auf die gesamten Daten. Mit den FACT-Tools ist man in der Lage, so eine *Feature-Extraktion* durchzuführen.

Die FACT-Tools bieten allerdings nicht nur diese Verarbeitungsschritte an, sondern können, je nach Analyseaufgabe, auch verschiedene andere Vorverarbeitungsschritte durchführen [22]. Ist die Datenvorverarbeitung abgeschlossen, kann mit der eigentlichen Datenanalyse begonnen werden.

11.1.3 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) Abschnitt 10.3 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.

Der Ablauf der Analysekette wird in Abbildung 11.1 veranschaulicht.

Abbildung 11.1: Analysekette

11.2 Grenzen von `streams`

von Mohamed Asmi

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. Auch 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 Parallelisieren 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.

Der Grund dafür ist, dass man die Ergebnisse von verschiedenens Einsätzen vergleichen kann.

Teil IV

Architektur und Umsetzung

Kapitel 12

Komponenten und Architektur

von Karl Stelzner

Bei Betrachtung der zu analysierenden Daten (Kapitel 10) wird deutlich, dass zur Umsetzung der in Kapitel 9 spezifizierten Ziele ein Big Data-System benötigt wird. Mehrere Eigenschaften von Big Data (vgl. Kapitel 3) 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 Kapitel 10 gesehen, werden von verschiedensten Sensoren Daten gesammelt, die anschließend in der Analyse kombiniert werden müssen.

Unser System basiert daher auf der in Kapitel 4 vorgestellten Lambda-Architektur für Big-Data-Systeme. Eine Übersicht über die verwendeten Software-Komponenten ist in Abbildung 12.1 dargestellt.

Den Kern des Systems bildet ein Apache Hadoop Cluster (vgl. Abschnitt 5.1). Dieser bietet zum einen das verteilte Dateisystem HDFS, mit dem große Datenmengen redundant und effizient abrufbar gespeichert werden können. Auf Grund dieser Eigenschaften wird es von uns zur Ablage der Rohdaten, also der in Kapitel 10 beschriebenen FITS-Dateien, verwendet. Um diese Daten und etwaige Zwischenergebnisse allerdings durchsuchbar zu machen, müssen sie indexiert werden. Hierfür verwenden wir verschiedene Datenbanksysteme. Da die genaue Ausprägung der zu speichernden Daten und der zu erwartenden Anfragen noch unklar ist, haben wir uns nicht auf ein System festgelegt, sondern verschiedene Lösungen implementiert. Diese werden in Kapitel 13 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. Für die Batchverarbeitung verwenden wir das Cluster Computing Framework Apache Spark, welches es erlaubt, verteilte Datensätze über den Hadoop Cluster zu verarbeiten (vgl. Abschnitt 5.2).

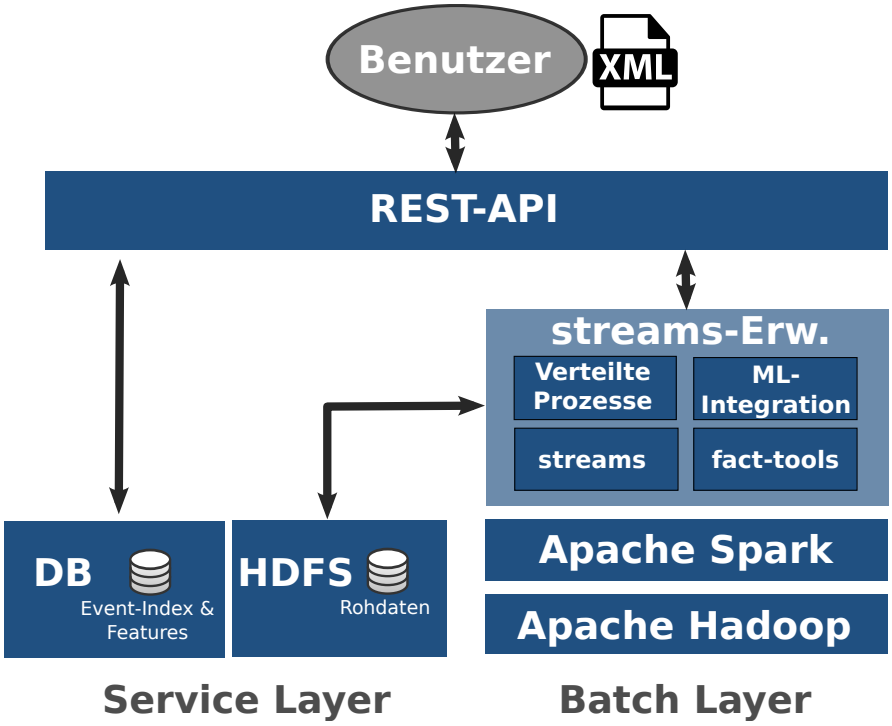


Abbildung 12.1: Überblick über die verwendeten Software-Komponenten

Um die verteilte Ausführung möglichst vieler Analyseaufgaben zu ermöglichen, erweitern wir das streams-Framework (vgl. Abschnitt 6.4) 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 Unterabschnitt 11.1.2) 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. 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 Teleskopdaten (vgl. Unterabschnitt 11.1.3) über dieselbe Schnittstelle spezifiziert werden kann. Näheres zur Implementierung und zu den Änderungen an der XML-Schnittstelle wird in Kapitel 15 erläutert.

Um dem Benutzer eine einheitliche Schnittstelle zu unserem System zu bieten, verwenden wir eine REST-API (vgl. Abschnitt 7.2). Diese versteckt einerseits die unterschiedlichen Anfragesprachen der Datenbanksysteme hinter einer gemeinsamen Schnittstelle und erlaubt es dem Benutzer andererseits, Anfragen über ein Webinterface zu stellen. Details zur Implementierung der API werden in Kapitel 14 beschrieben.

Ein Speed-Layer zur Verarbeitung von Daten in Echtzeit ist von uns bisher noch nicht

umgesetzt worden. Denkbar wäre hierzu die Nutzung einer der in Kapitel 6 vorgestellten Technologien. Dies wird ein Fokus unserer Arbeit im kommenden Semester sein.

Kapitel 13

Indexierung der Rohdaten

von Karl Stelzner

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. Kapitel 10). 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 mit Hilfe 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 verwendeten 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.

13.1 MongoDB

von Christian Pfeiffer

Das Ziel, einen Index für die Rohdaten zu erstellen, kann in MongoDB (siehe Unterabschnitt 7.1.1) 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 die selbe 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 mit Hilfe 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 Abschnitt 10.5 erläuterte Strategie zum Finden des passenden Messwertes für ein Event eingesetzt. Weitere Experimente zur Performance des Indexierungsvorgangs sowie zur Suchgeschwindigkeit innerhalb der Indexdatenstrukturen folgen im zweiten PG-Semester.

13.2 Elasticsearch

von Lea Schönberger

Um die Performanz verschiedener Datenbanken hinsichtlich des Anwendungsfalles dieser Projektgruppe gegeneinander abwägen zu können, wird 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* wird Elasticsearch momentan lediglich auf einem einzigen Rechenknoten betrieben, dies soll sich jedoch künftig ändern, sodass auf jedem verfügbaren Knoten des Clusters des Sonderforschungsbereiches 876 ein Elasticsearch-Node betrieben wird.

13.3 PostgreSQL

von Karl Stelzner

Als dritte mögliche Lösung haben wir ein PostgreSQL System aufgesetzt, also ein herkömmliches relationales Datenbankmanagementsystem (vgl. Unterabschnitt 7.1.4). 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. Unterabschnitt 7.1.4). Eine andere Möglichkeit ist der Einsatz des JSON-Datentyps, den PostgreSQL anbietet. Neue Features könnten dann einfach in bestehende Tabellenzeilen eingefügt werden. Wie performant und skalierbar diese Lösung ist, muss noch getestet werden.

Als erstes Experiment haben wir eine Tabelle für die Metadaten erstellt, und diese mit dem selben Inhalt wie die MongoDB befüllt. Die Ergebnisse des Leistungsvergleichs finden sich in Abschnitt 17.1.

Kapitel 14

Umsetzung der RESTful API

14.1 Design

von Alexander Schieweck

Zur Umsetzung der RESTful API (vgl. Abschnitt 7.2) 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. [68]

14.1.1 Endpunkte

von Alexander Bainczyk

Die Endpunkte der REST API sind so gewählt, dass der Zugriff auf indizierte Daten in den in der PG genutzten Datenbanken (MongoDB, Elasticsearch und PostgreSQL) einheitlich verläuft. Wie bereits im vorigen Abschnitt beschrieben, sind zum Zeitpunkt des Schreibens noch keine Endpunkte für die PostgreSQL Datenbank realisiert.

URL	GET-Parameter
GET /api/mg/events	format, filter
GET /api/es/events	format, filter

Tabelle 14.1: Schnittstellen der REST API für Metadaten

Die in Tabelle 14.1 aufgelisteten Schnittstellen sind für den Zugriff auf Metadaten von Events konzipiert, wobei die Abkürzung *mg* für die MongoDB- und *es* für die Elasticsearch-Datenbank steht. Die Angabe der GET-Parameter ist optional. Hierbei kann über *format* das Rückgabeformat einer Antwort bestimmt werden (s. Unterabschnitt 14.1.2). Über den Parameter *filter* lässt sich ein Filterausdruck übergeben, mit dem die Metadaten selektiert werden können (s. Unterabschnitt 14.2.2).

```
[
  {
    "EVENT_NUM": "4",
    "TRIGGER_NUM": "4",
    "NIGHT": "20130921",
    ...
  },
  {
    "EVENT_NUM": "5",
    "TRIGGER_NUM": "4",
    "NIGHT": "20130921",
    ...
  },
  ...
]
```

(a) JSON

```
[
  {
    "path": ".../hdfs/fact/raw/2013/08/21/....fits.gz",
    "eventNums": [20, 22, 24, 50, ...]"
  },
  {
    "path": ".../hdfs/fact/raw/2013/09/06/....fits.gz",
    "eventNums": [2, 22, 120, 121, ...]"
  },
  ...
]
```

(b) Minimal

Abbildung 14.1: Die Rückgabeformate der REST API

14.1.2 Rückgabeformate

von Alexander Bainczyk

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/mg/events/?filter=[...]&**format=json**

Die Antwort würde in diesem Fall aussehen, wie in Abbildung 14.1a 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 Abbildung 14.1b 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/mg/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.

14.1.3 Dokumentation

von Alexander Schieweck

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[1] entschieden. Dort wurde eine Spezifikation, die mittlerweile von der Open API Initative[2] betreut wird, entwickelt, mit der sich RESTful APIs mithilfe von JSON beschreiben lassen[3]. Rund um diese Dokumentation sind unterschiedliche Tools entstanden[4], z.B. ein Text-Editor, um das JSON, welches die API beschreibt, einfacher bearbeiten zu können[5]. Noch hilfreicher ist jedoch die Swagger UI[6], 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.

14.2 Implementierung

14.2.1 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 Elasticsearch und MongoDB ausgewählt, welche im Folgenden näher erläutert werden.

[1] http://swagger.io
[2] https://openapis.org/
[3] https://github.com/OAI/OpenAPI-Specification
[4] http://swagger.io/open-source-integrations
[5] http://swagger.io/swagger-editor
[6] http://swagger.io/swagger-ui

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 14.1: 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 14.2: 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. Das vollständige Funktionsspektrum kann in der Online-Dokumentation[7] nachgelesen werden.

- 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.

14.2.2 Filterung

von Dennis Gaidel

Der Ansatz der Implementierung einer Schnittstelle mit Hilfe von REST Ressourcen basiert 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

[7] http://projects.spring.io/spring-boot/

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 14.3 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.

```
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 14.3: 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 14.4 dargestellt wird.

```
1 { $and: [
2     { $or: [
3         { event_num: { $gte: 5, $lte: 10 }},
4         { event_num: { $gte: 50, $lte: 100 }}
5     ]},
6     {
7       trigger_num: { $gt: 10 }
8     }
9 ]}
```

Listing 14.4: 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 Unterabschnitt 14.1.1 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 Unterabschnitt 14.2.1 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* [8] 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.

[8] `https://github.com/querydsl/querydsl`

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 14.5).

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

Listing 14.5: 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 14.6 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 14.6: 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 14.5).

Der Ausdruck muss also zur Laufzeit in ausführbaren Java-Code übersetzt werden, was mittels der Ausdruckssprache *MVEL* [9] erreicht wird. Diese Ausdruckssprache ist an die

[9] `https://github.com/mvel/mvel`

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 14.7 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 14.7: Evaluation der Anfrage

Nach der Erzeugung des `Predicate` Objekts kann dieses an das entsprechende Repository übergeben werden, wie es beispielsweise in Listing 14.8 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 14.8: 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 14.9 zu entnehmen ist. Durch diesen Mechanismus garantiert die Einbindung des `QueryDslPredicateExecutor`s, 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 14.9: 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 vereinheitlich 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 *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.

Kapitel 15

Erweiterung der Streams-Architektur

von Mirko Bunse, David Sturm, Christian Pfeiffer

Analysten der FACT-Daten verwenden zurzeit das Streams-Framework [20]. Es ermöglicht die Spezifikation einer Streaming-Applikation durch ein XML-Dokument (für weitere Informationen siehe Abschnitt 6.4). Da Streams den Datenanalysten bereits bekannt ist, empfiehlt es sich, für die PG, auf Streams aufzubauen und die Ergebnisse als Erweiterung des Frameworks zu konzipieren.

Mit Streams-Storm existiert bereits eine BigData-Erweiterung für Streams. Das dort verwendete Apache Storm [10] stellt eine Infrastruktur für Streaming-Applikationen von großen Datenvolumen in Clustern dar. Wir verwenden für unsere Erweiterung Apache Spark [11], welches in Hinblick auf die Geschwindigkeit ein prominenter Konkurrent von Storm ist. Apache Spark führt seine Applikationen im Cluster verteilt als Batch-Jobs aus, wobei hohe Ausführungsgeschwindigkeiten durch Vorhaltung der Daten im Hauptspeicher erreicht werden. Spark-Streaming [12] ermöglicht darüber hinaus Streaming-Applikationen, indem die Daten in Ketten von Mini-Batches analysiert werden. Für weitere Informationen zu Spark und Spark-Streaming siehe auch Abschnitt 5.2 und Abschnitt 6.3.

Durch die Entwicklung einer Spark-Erweiterung für das Streams-Framework wollen wir feststellen, wie gut sich die beiden Ansätze von Spark zur Analyse der FACT-Daten eignen. Gibt es möglicherweise unterschiedliche Analysen, die durch jeweils andere Ansätze besser abgedeckt werden?

Wir erweitern Streams um zwei Komponenten, die die Möglichkeiten von Spark ausnutzen. Zunächst wollen wir eine verteilte Ausführung von Streams-Prozessen ermöglichen. Bisher kann ein solcher Prozess, der mehrere unabhängige Eingabeströme verarbeiten soll, dies lediglich sequentiell tun. Es bietet sich an, diese Eingabeströme auf die Rechner des Clusters zu verteilen. Zu diesem Zweck erstellen wir einen `DistributedProcess` (siehe Abschnitt 15.1), welcher Spark zur Synchronisation der Ergebnisse verwendet. Er gestaltet die verteilte Ausführung für den Autor der XML-Konfiguration weitestgehend transparent (lediglich das `distributedProcess`-Tag ist statt des Streams-Tags `process` zu verwenden).

Die zweite Erweiterung von Streams zielt darauf ab, die Methoden der Machine-Learning-Bibliothek Spark ML [9] verfügbar zu machen. Diese sind bereits für die Ausführung im Rechencluster ausgelegt und ermöglichen unter anderem das Trainieren von Vorhersagemodellen (für weitere Informationen zu MLlib siehe Unterabschnitt 5.2.3). Zur Anbindung an MLlib stellen wir gleich mehrere neue Elemente (`input`, `task`, ...) für XML-Spezifikationen bereit (siehe Abschnitt 15.2). Durch diese Elemente streben wir die syntaktische Trennung der Batch-Algorithmen in MLlib von den bestehenden Elementen mit Streaming-Semantik an. Unsere MLlib-Elemente können selbstverständlich trotzdem mit anderen Streams-Elementen kombiniert werden.

Die Kombination der beiden Erweiterungen erlaubt es uns, innerhalb einer einzigen XML-Konfiguration zuerst Vorverarbeitungsschritte mit verteilten Prozessen als Streams zu realisieren, als auch mit den resultierenden Features ein Modell mit MLlib einzupassen. Auf diese Weise wird eine Basis geschaffen, mit der später auch komplexe Abläufe modelliert werden können.

15.1 Verteilte Streams-Prozesse mit Spark

von Mirko Bunse

Das für unsere Erweiterung verwendete Apache Spark verteilt Datenverarbeitung in Rechenclustern. Dieses Konzept skaliert sehr gut horizontal, d.h., die Performanz lässt sich durch Anbindung weiterer Cluster-Knoten steigern. Da horizontale Skalierbarkeit eine Schlüsseleigenschaft von BigData-Anwendungen darstellt (siehe Abschnitt 3.3), wollen wir die Verarbeitung der Daten im Streams-Framework geeignet mit Spark verteilen.

15.1.1 Nebenläufigkeit der Verarbeitung

von Mirko Bunse

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 Abschnitt 6.4).

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

Diese Erkenntnisse beschränken sich nicht auf Apache Spark. Wir werden Spark aber verwenden, um den hier vorgestellten Ansatz der Verteilung umzusetzen (siehe Unterabschnitt 15.1.4).

15.1.2 XML-Spezifikation verteilter Prozesse

von Mirko Bunse

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. Das einzig neue Tag zur Verteilung von Streams-Prozessen ist `distributedProcess`, was sich von Default-Prozessen durch die Verteilung der Verarbeitung auf Worker-Knoten im Cluster unterscheidet.

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. Unterabschnitt 15.1.1). 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 15.1 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!

Da es im BigData-Umfeld vorkommt, dass ausgesprochen viele Streams (z.B. zur Verarbeitung hunderter .fits-Dateien) zu erzeugen sind, haben wir einen MultiStream-Generator konzipiert, der eine Menge von Streams als MultiStream erzeugt. Damit müssen nicht alle inneren Streams mühselig aufgelistet werden. Stattdessen erlaubt eine Regular Expression über Datei-Pfade, alle benötigten Streams in einer Zeile zu spezifizieren (siehe Unterabschnitt 15.1.5).

```
<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 15.1: Beispiel-XML für die Nutzung eines DistributedProcess

15.1.3 Verarbeitung der XML-Spezifikation

von Mirko Bunse

Damit das neue Tag `distributedProcess` verwendet werden kann, mussten wir einen Handler für XML-Elemente dieses Tags schreiben. Der bestehende Parser erzeugt Objekte solcher Elemente, welche dann von unserem neuen Handler verarbeitet werden können. Der Handler hat eine Factory aufzurufen, die verteilte Prozesse erzeugt.

Für die Implementierung der Factory verteilter Prozesse 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 des Typs DistributedProcess 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.

15.1.4 Ansatz unter der Spark Core-Engine

von Mirko Bunse

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.

Verteilung von Streams-Prozessen

von Mohamed Asmi

Nach dem ersten Experimenten unter der Verwendung von Spark wurde festgestellt, dass die Datenströme sequenziell verarbeitet wurden. Jedoch wurde jeder eingehende Datenstrom parallel verarbeitet. Dadurch ist der gewünschte Performanzgrad nicht erreicht worden. Auf diesem Grund wurde die Idee von der Verteilung der gesamten Datenstromverarbeitung umgesetzt.

Für die Realisierung wurde die Klasse `DistributedProcess` implementiert. Sie sorgt dafür, eine erhöhte Performanz zu schaffen. Ein `DistributedProcess`-Objekt ist in der Lage, zu erkennen, ob aus einer Datenstromquelle einer oder mehrere Datenströme kommen. In dem Fall, in dem nur ein Datenstrom verarbeitet wird, wird ein Standardprozess vom streams-Framework [20] durchgeführt. Dieser Prozess wird auf einem einzelnen Arbeitsknoten (auch Worker genannt) ausgeführt.

Anderseits, wenn mehrere Datenströme für die Verarbeitung zu Verfügung stehen, wird die verteilte Verarbeitung ausgelöst. Die Verarbeitung der einzelnen Datenströme wird auf die Worker verteilt. Wie die einzelnen konkret arbeiten wird in dem nächsten Abschnitt untersucht.

Bei der Verteilung werden nicht die Datenströme verteilt, sondern nur deren IDs. Damit müssen nicht die Daten immer im Netzwerk zwischen den Knoten verschickt werden. Die Datenstrom-IDs werden in RDDs verpackt und für die einzelnen Workern weitergegeben. Darüber hinaus wird ein `DistributedProcessContext`-Objekt an alle Workern durch eine Spark-Broadcast-Variable [11] geschickt. Der `DistributedProcessContext` besteht aus dem XML-Dokument und der Prozess-ID. Dadurch hat jeder Worker lokal die benötigten Ressourcen für die Ausführung des Prozesses. Um das gesamte Ergebnis am Ende der Verarbeitung zusammenzufassen, wird ein `Accumulable`-Objekt von Spark verwendet. Es dient dazu, dass die Teilergebnisse aus den einzelnen Workern gesammelt und gebunden werden. Daher liegt am Ende das gesamte Ergebnis im Output auf dem Driver. Der Ablauf wird in Abbildung 15.1 veranschaulicht.

Instanziierung des `streams`-Frameworks in den Workern

von Karl Stelzner

Die Aufgabe eines jeden Workers besteht nun darin, den Prozess für eine Teilmenge der Daten auszuführen, nämlich für den Eingabestream, dessen ID übergeben wurde. Wie oben gesehen verfügt der Worker über alle notwendigen Informationen, nämlich das XML-Dokument (als Objektbaum), die Prozess-ID, und die Stream-ID. Um die tatsächliche Ausführung umzusetzen, gibt es allerdings mehrere Alternativen.

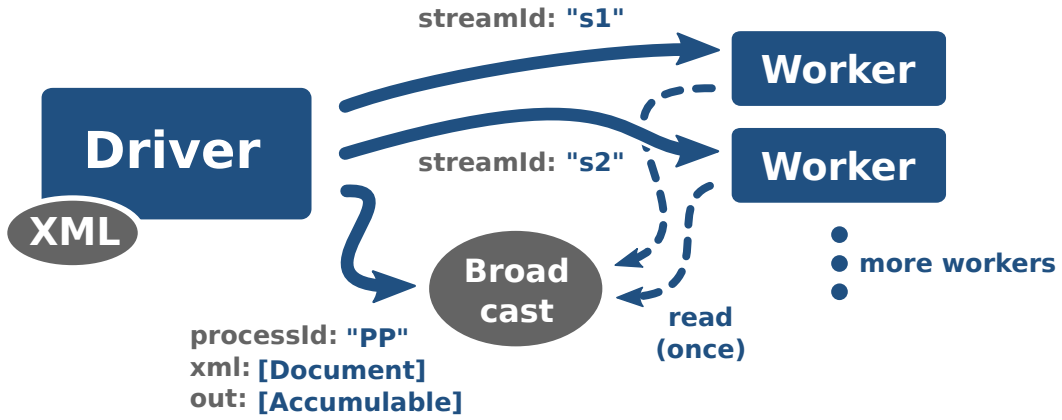


Abbildung 15.1: Verteilung eines Streams-Prozesses

Klar ist zunächst, dass es vorteilhaft ist, die `streams`-Klassen zur Interpretation 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. Dafür gibt es im Wesentlichen drei Möglichkeiten:

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 Worker ein eigenes XML-Dokument erstellt werden muss, was unter Umständen zu viel 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 parallele Prozess hier wie ein regulärer behandelt wird. Um die von uns gewünschte Semantik zu gewährleisten, lässt sich der Worker dann über

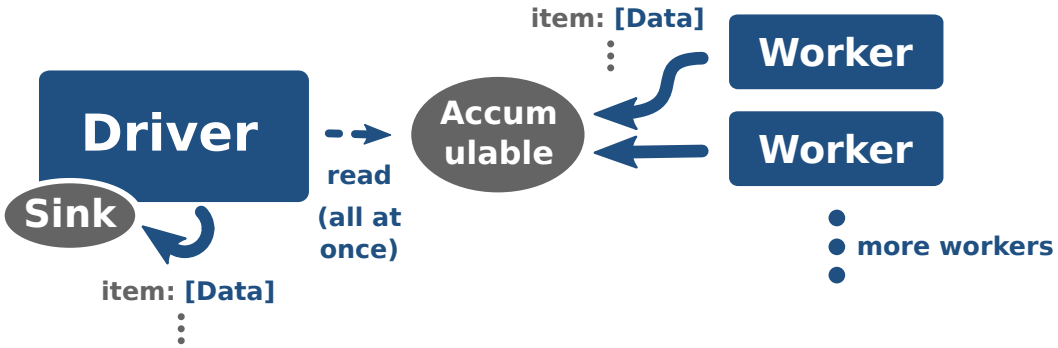


Abbildung 15.2: Zusammenfassung der Teilergebnisse per Accumulable

die `getProcesses()` Methode die Liste der erzeugten Prozesse geben und entfernt daraus alle bis auf den gewünschten. Dessen Eingabe wird dann von dem im XML-Dokument spezifizierten Multistream auf den gewünschten Substream geändert. Dies sorgt dafür, dass die gewünschte Semantik durch einen schlichten `execute()` Aufruf beim ProcessContainer erreicht wird.

Zusammenfassung der Teilergebnisse

von Karl Stelzner

Der vorherige Abschnitt erläutert, wie wir dafür gesorgt haben, dass auf jedem Worker eine Instanz des verteilten Prozesses mit korrekter Eingabe ausgeführt wird. Noch offen ist die Frage, wie der Teilergebnisse der Worker an den Driver zurückgeschickt werden.

Eine Möglichkeit, die Spark hierfür bietet, ist die Klasse Accumulable. Diese erlaubt es, Daten der Workernodes in einer gemeinsamen Datenstruktur zu aggregieren. Anschließend kann dann die gesamte Datenmenge auf Seite des Drivers abgerufen werden. Um die Verbindung zwischen dieser Funktionalität und der streams-Logik herzustellen, haben wir die Klasse `AccumulableListSink` geschrieben, die von Accumulable erbt und gleichzeitig das Sink-Interface von Streams implementiert. Werden Daten über die Sink-Methode `write` geschrieben, leitet diese Klasse sie einfach über die `add` Methode an das Accumulable weiter.

Ein Objekt dieses Typs wird den Workern via Broadcast zu Verfügung gestellt. Dort wird es über das im vorigen Abschnitt erläuterte Verfahren als Ausgabeobjekt des auszuführenden Prozesses gesetzt. Dies sorgt dafür, dass die auf dem Worker laufende `streams`-Instanz ihre Ergebnisse automatisch in das als Senke getarnte Accumulable schreibt.

Nachdem alle Worker ihre Arbeit fertiggestellt haben, kann der Driver die Gesamtheit der Ergebnisse als verkettete Liste von Datenobjekten aus dem Accumulable auslesen. Dieser Ablauf wird durch Abbildung 15.2 verdeutlicht.

Eine alternative Vorgehensweise besteht darin, die Worker über die `flatmap`-Methode aufzurufen. Diese sorgt dafür, dass deren Teilergebnisse automatisch in einer Ergebnis-RDD

zusammengefasst werden. Deren Inhalt kann dann durch den Driver über die `collect`-Methode aggregiert werden. Beide Vorgehensweisen führen im Wesentlichen zum gleichen Verhalten und teilen die selben Schwächen (vgl. Abschnitt 16.2). Der Unterschied liegt lediglich darin, dass der Einsatz von Accumulables eine etwas flexiblere und allgemeinere Zuweisung der Ergebnisse erlaubt, und daher von uns zuerst umgesetzt wurde. Die Nutzung von Ergebnis-RDDs ist dagegen schlanker in der Umsetzung und entspricht eher der Spark-Designphilosophie, weshalb wir sie vermutlich für das Endprodukt bevorzugen werden.

15.1.5 MultiStream-Generatoren

von Mohamed Asmi

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 15.2).

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

Listing 15.2: Beispiel Multistream Eingabe

Der Performanzgewinn wird in Abschnitt 16.1 diskutiert.

15.2 MLLib in Streams

von Carolin Wiethoff

Zur Arbeit des Endprodukts 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, beziehungsweise die Methoden des darin enthaltenen und von uns favorisierten Paketes ML (siehe Abschnitt 5.2.3), 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 den im vorherigen Abschnitt 15.1 vorgestellten *DistributedProcess* durchgeführt. Mit unserer Erweiterung soll es jedoch auch möglich sein, MLlib-Methoden zu nutzen, wenn dies gewünscht ist. 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.

15.2.1 XML-Spezifikation von input

von Christian Pfeiffer

Ein `input`-Tag dient dazu, eine Datenquelle zu spezifizieren, die einen DataFrame (siehe Unterabschnitt 5.2.2) 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 eines Containers stehen. Ein Beispiel hierfür findet sich in Listing 15.3.

15.2.2 XML-Spezifikation von task & operator

von David Sturm

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

15.2.3 XML-Spezifikation von pipeline

von Michael May, Lili Xu

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 einer Task definiert und kann dann durch Spezifizierens eines Namens im weiteren Verlauf verwendet werden (Listing 15.3).

```
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 15.4: Beispiel-XML einer reduzierten Pipeline innerhalb einer Task

Listing 15.4 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.

15.2.4 XML-Spezifikation von stages

von Carolin Wiethoff

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 15.5: Beispiel-XML für die Verwendung des estimator-Tags

In Listing 15.5 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 15.6: Beispiel-XML für die Verwendung des transformer-Tags

In Listing 15.6 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 weiter verarbeiteten *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 15.7: Beispiel-XML für die Verwendung der estimator- und transformer-Tags innerhalb einer Pipeline

15.2.5 Umsetzung

von Michael May

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 *von Christian Pfeiffer*

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 *von Michael May*

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 15.3 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

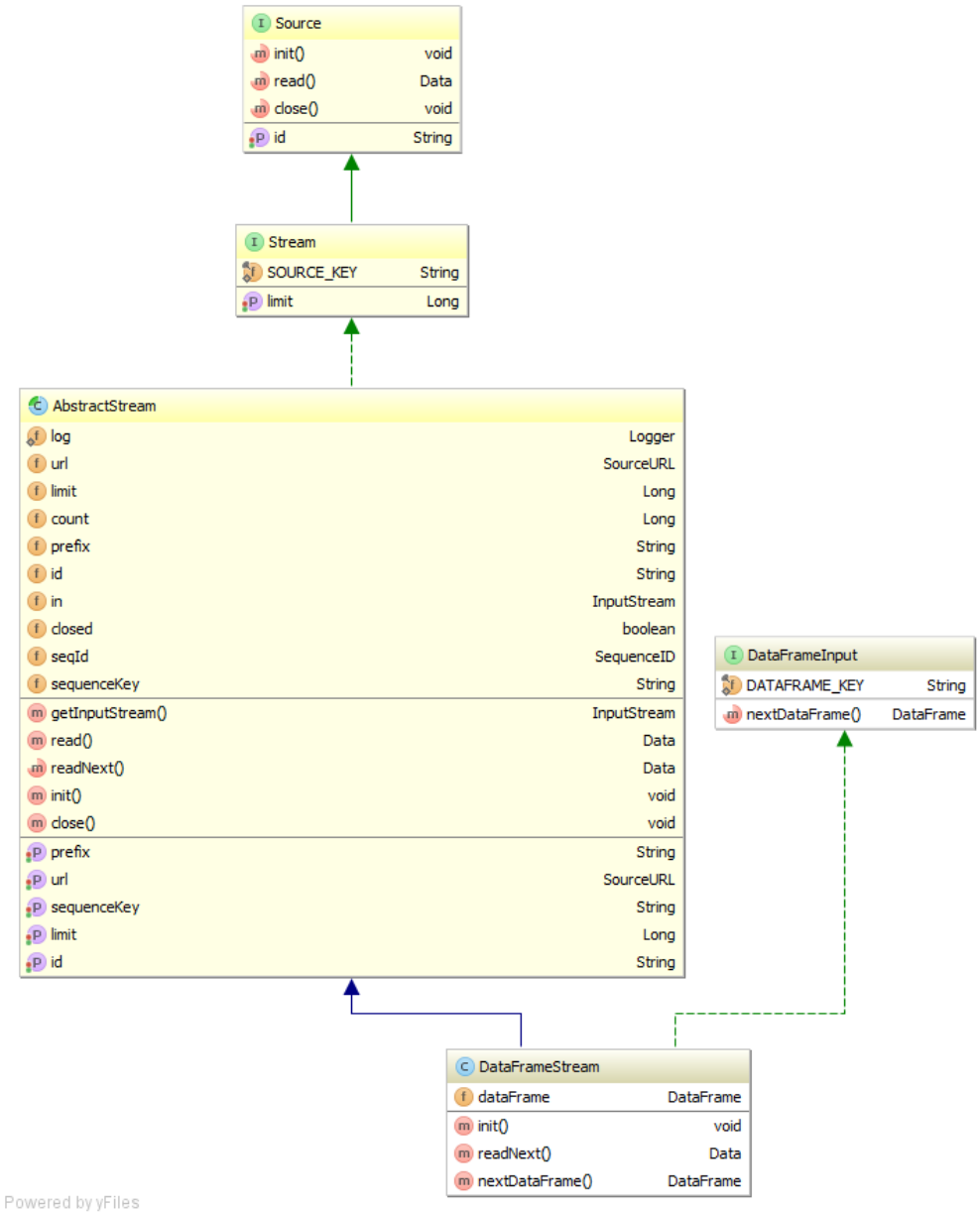


Abbildung 15.3: Klassendiagramm mit zugehörigen Klassen für `DataFrameInput` und `DataFrameStream`

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.

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 dieses 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 *von Michael May*

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. 15.4) und Verarbeitung (Abb. 15.5) 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 15.5 zeigt eine Übersicht der so erstellten Pipeline und Stage-Instanzen. Hierbei sei anzumerken, dass während der Entwicklung des Prototypen verschiedene Ansätze verfolgt wurden und auch die derzeitige Version einen *work-in-progress* darstellt.

Eine instantiierte Pipeline, wie beispielsweise eine `DefaultPipeline`, verarbeitet *DataFrame* Objekte, weswegen sie als Spezialisierung des `AbstractOperator` implementiert wur-

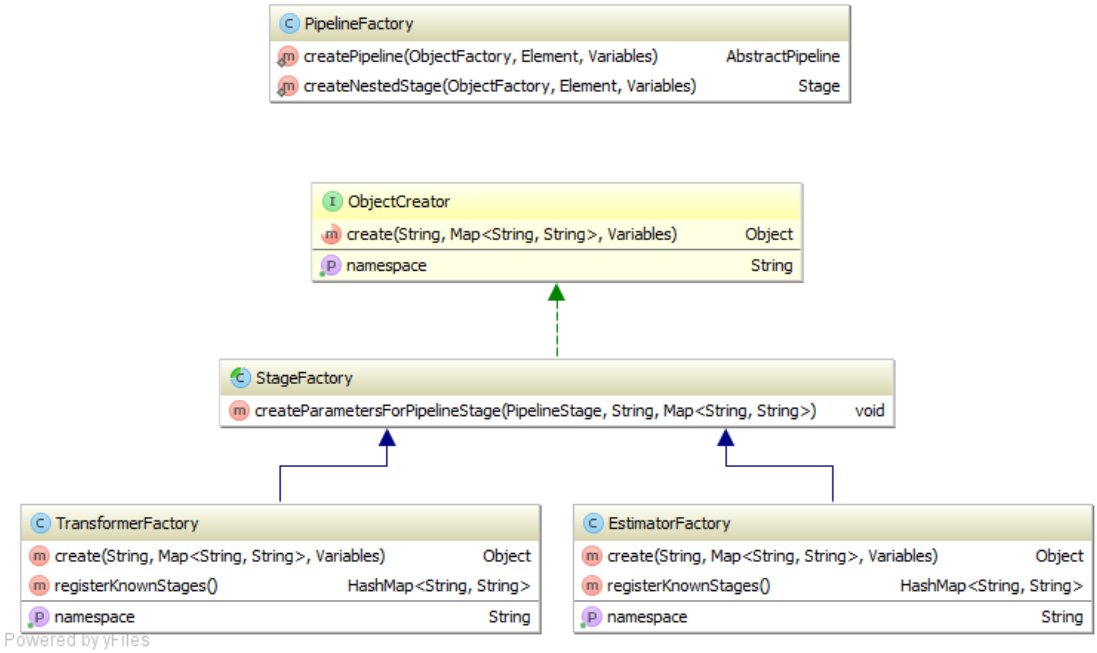


Abbildung 15.4: Übersicht der Klassen zuständig für die Erstellung von Pipelines und Stages

de. Da Pipelines spezialisierte Prozessoren sind, kann die `streams` Implementierung genutzt werden, um Daten zu verarbeiten, und im Fall der Pipeline Dataframes. In der `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.

In der aktuellen Version des Prototypen wurden Klassen für die Pipelinestages erstellt, welche als abstrakte Ebene zwischen den Spark-ML-Stages und `streams`-Stages. Allerdings findet sich derzeit kein Nutzen in dieser Abgrenzung, weswegen eine wünschenswertes Ziel für die nächste Version den Zweck einer solchen Ebene zu untersuchen.

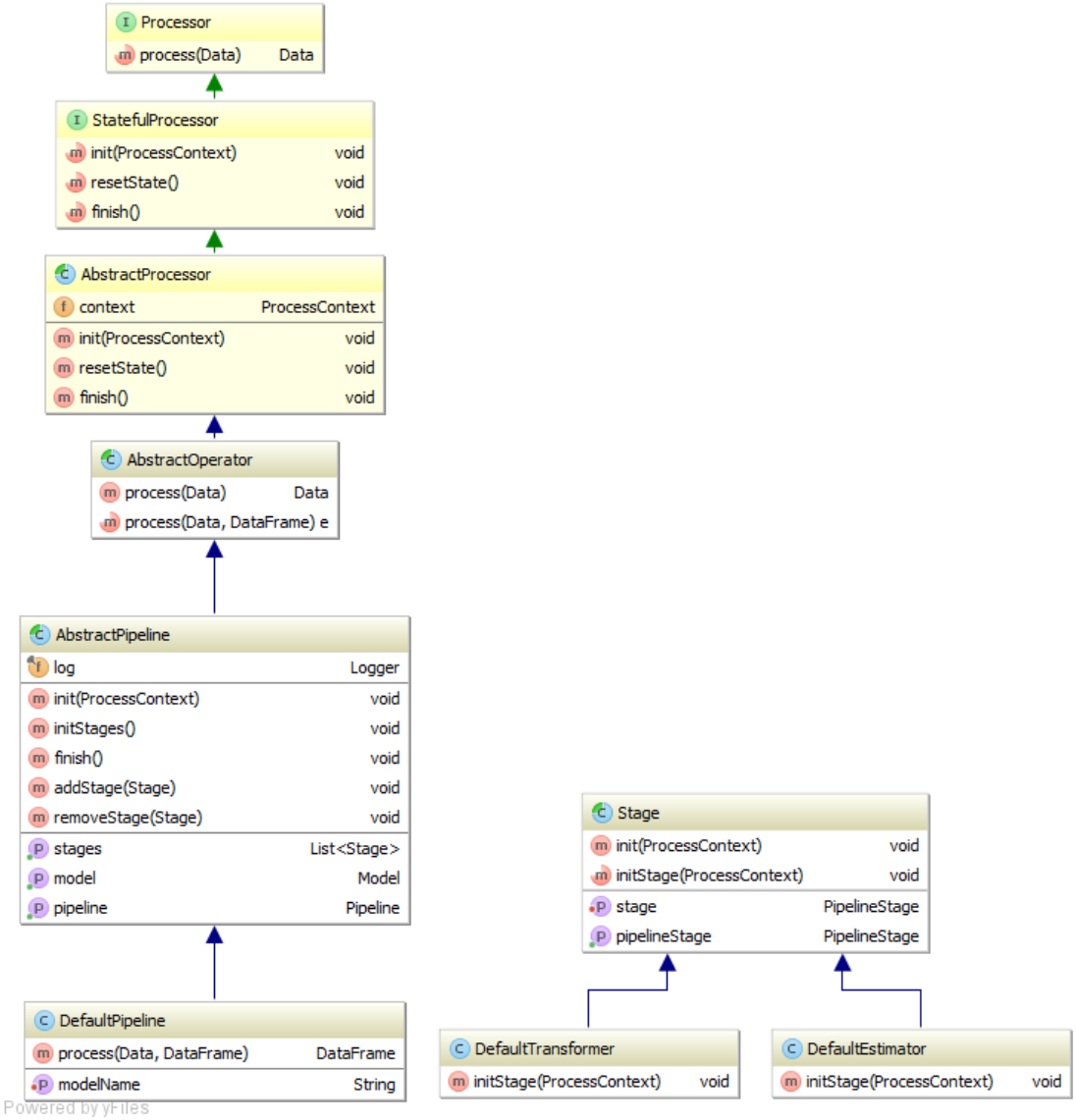


Abbildung 15.5: Übersicht der Klassen zuständig für die Verarbeitung von Pipelines und Stages

Teil V

Evaluation und Ausblick

Kapitel 16

Vergleich mit streams

von Mohamed Asmi

Im Rahmen dieser Projektgruppe (PG) wurde eine Spark-Erweiterung des *streams-Frameworks*[20] implementiert. Dadurch wird ein verteiltes Verarbeiten der Daten durch das *FACT-Tools* unterstützt. Durch die Eingabe von mehreren Datenquellen wird der `StreamsGenerator` angeschaltet. Der Generator baut die Datenströme ein und leitet sie für die verteilte Verarbeitung weiter. Dabei werden die Datenströme an verschiedene `worker` verschickt. Wenn hingegen keine verteilte Verarbeitung notwendig ist, wird ein Standard-Prozess ausgeführt.

Das Ziel der Erweiterung ist, eine bestimmte Performanz zu erreichen. Durch die verteilte Verarbeitung wird eine schnellere und effizientere Verarbeitung der Daten erzielt. Bei Big Data soll es ein Vorteil sein, da es immer große Datenmengen verwendet. Aber man kann nicht ausschließen, dass bei der Verarbeitung von großen Datenmengen andere Probleme auftreten können.

In diesem Kapitel wird der Performanzgewinn durch die Spark-Erweiterung betrachtet. Dafür werden Experimente durchgeführt, bei denen Datenmenge verschiedener Größe einmal mit dem *streams-Framework* und mit deren Spark-Erweiterung verarbeitet werden.

16.1 Performanzgewinn durch verteilte Prozesse

von Mohamed Asmi

Für die Experimente wird die MC-Analyse in Hinblick auf die Monte-Carlo-Daten betrachtet, bei der die Daten für die Anwendung eines maschinellen Lernverfahrens bereitgestellt werden. Dabei wird die Verarbeitungszeit von Events untersucht. Ein Event ist ein Dataitem, das von dem *streams-Framework*geliefert wird. Für die Zeitmessung wurde eine Sinke implementiert. Der `PerformanceSink` ist in der Lage, die Verarbeitungszeit eines Datenitems bzw. Events zu messen. Die Messergebnisse werden in eine Datei geschrieben. Alle Tests werden auf dem Hadoop-Cluster des Sonderforschungsbereiches 876 durchgeführt.

Bei den Experimenten werden zwei Fälle betrachtet. Im ersten Fall wird nur ein einzelner Datenstrom betrachtet, d.h., die Daten kommen nur aus einer Datei. Der Grund für diese

Auswahl ist, dass gezeigt werden soll, dass, die `streams-Erweiterung` in der Lage ist, automatisch zu erkennen, ob mehrere oder nur eine Datenquelle betrachtet wird. Außerdem wird als Hauptgrund die Untersuchung der Verarbeitungszeit der Events, unter die Lupe genommen.

Für die Ausführung der Experimente werden insgesamt 6GB Arbeitsspeicher verwendet. Also in dem Fall, in dem verteilte Prozesse betrachtet werden, werden 3 Cluster Knoten mit jeweils 2GB verwendet. In dem anderen Fall wird 6GB Arbeitsspeicher für den Master-Knoten zur Verfügung gestellt.

Zuerst wird die `MC-Analysis` nur auf einem Datenstrom betrachtet. Die Ergebnisse sind in Abbildung 16.1 zu sehen.

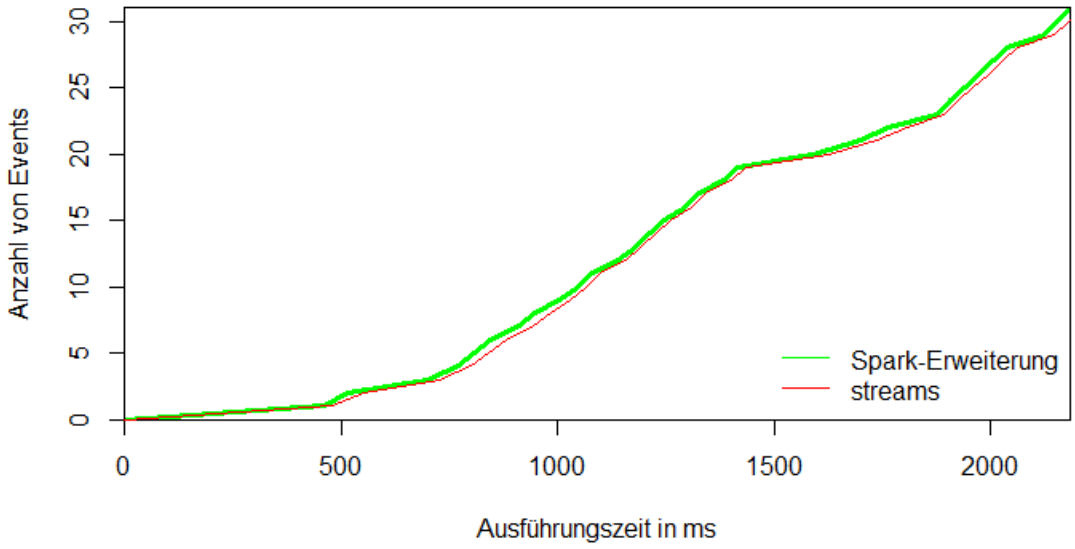


Abbildung 16.1: Vergleich von der Spark-Erweiterunf mit streams (einzel Stream)

In diesem Fall ist keine verteilte Verarbeitung notwendig. bei der Spark-Erweiterung wird der Standard-Prozess des `streams-Frameworks` durchgeführt. Das erklärt die gleichen gemessenen Ausführungszeiten, die in Abbildung 16.1 zu sehen sind.

Jetzt werden mehrere Dateien als Datenquellen genutzt. Verwendet werden 200 Dateien der `proton klaus`, die aus der `Monte-Carlo-Simulation` entstanden sind. Die Ergebnisse sind in Abbildung 16.2 zu sehen.

Bei dem Bereitstellen von mehreren Datenströmen wird die verteilte Verarbeitung angeschaltet. Man kann bei der Spark-Erweiterung in Abbildung 16.2 beobachten, dass am Anfang nichts passiert und dann auf einmal alle Daten in einer konstanten Zeit verarbeitet

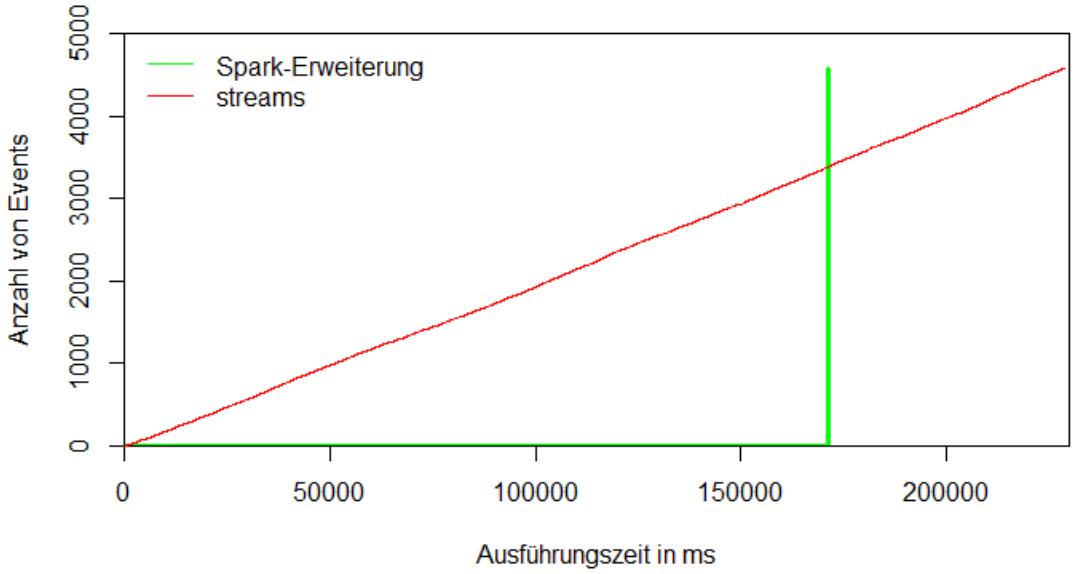


Abbildung 16.2: Vergleich von der Spark-Erweiterunf mit streams (Multistream)

wurden. Dagegen steigt die Ausführungszeit linear mit der Anzahl der Events. Insgesamt wurde bei der Verarbeitung von ungefähr 4600 Events 228 Sekunden mit dem *streams-Framework* gebraucht und nur 170 Sekunden bei dem Anwenden der Spark-Erweiterung. Je mehr Daten verarbeitet werden, desto größer ist der Performanzgewinn. Bei der Verarbeitung von ungefähr 22800 Events wurde fast 8 Minuten mit der Spark-Erweiterung gebraucht. Dabei wurden aber insgesamt 24GB Arbeitsspeicher genutzt.

Bei der in der PG entwickelten Spark-Erweiterung des `streams-Framework` ist ein Performanzgewinn durch die verteilte Verarbeitung deutlich zu erkennen. Dieser Erfolg ist aber auch von ein paar Problemen begleitet. Diese werden in dem nächsten Abschnitt diskutiert. Darüber hinaus besteht der Cluster nur aus zwei Servern und die sechs Maschinen, die man sehen kann, sind nur virtuell. Aus diesem Grund entsteht wenig Netzwerkverkehr, denn da das HDFS drei Replikationen verwendet, hat meistens einer der beiden Server die gewünschte Datei lokal liegen. Deshalb entsteht wenig Netzwerkverkehr, der die Ausführungszeiten beeinflussen kann. Aus diesem Grund könnte es sein, dass bei der Ausführung auf einem Cluster mit realen Knoten längere Verarbeitungszeiten entstehen könnten.

16.2 Probleme verteilter Prozesse unter Spark

von Karl Stelzner

Der Einsatz von Spark zur Implementierung verteilter Prozesse geht mit einigen konzeptionellen Problemen einher. Diese rühren vor allem daher, dass Spark auf Batch-Verarbeitung

basiert, während dem `streams`-Framework Datenströme aus einzelnen Datenpunkten zugrunde liegen. Dies hat zum Einen zur Folge, dass verteilte Prozesse aktuell nur Eingaben verarbeiten können, die zu Beginn der Ausführung schon vollständig zur Verfügung stehen. Insbesondere ist es also nicht möglich, die Ausgabe eines anderen Prozesses verteilt weiterzuverarbeiten. In der Zukunft ließe sich dies allerdings durch Spark-Streaming umsetzen, da damit eintreffende Datenpunkte in regelmäßigen Zeitabständen zu Mini-Batches verarbeitet werden können.

Zum Anderen hat Spark den Nachteil, dass Ergebnisse erst dann verfügbar werden, wenn ein Batch vollständig abgearbeitet wurde. Dies ist sowohl bei der Nutzung von Akkumulatoren als auch bei der Verwendung von RDDs (via `RDD.collect`) der Fall. Dies verletzt nicht nur die Design-Philosophie des `streams`-Frameworks, sondern kann bei sehr großen Datenmengen auch dafür sorgen, dass das Driverprogramm unter der gebündelten Datenlast zusammenbricht.

Eine naive Möglichkeit, dieses Problem zu adressieren, ist die manuelle Partitionierung des Eingabe-Multistreams in kleinere Mini-Batches, die sequentiell abgearbeitet werden. Dies ist mit zwei Problemen verbunden: Zum Einen muss sichergestellt sein, dass diese Mini-Batches groß genug sind, um noch sinnvoll verteilt werden zu können. Zum Anderen kann es passieren, dass Workernodes auf einander warten müssen, wenn Streams unterschiedlich lang sind. Das liegt daran, dass der nächste Batch immer erst gestartet werden kann, wenn alle Streams des vorherigen abgearbeitet sind. Spark Streaming hilft in diesem Fall nicht weiter, da die Mini-Batches hier ebenfalls rein sequentiell verarbeitet werden.

Auch wenn die Rückgabe so in kleinere Batches zerteilt werden kann, entspricht dies immer noch nicht der `streams`-Semantik. Eleganter und effizienter wäre es, wenn die Workernodes ihre Ergebnisse als kontinuierlichen Datenstrom zurückgeben würden. Eine solche Funktionalität ist in Spark aber nicht vorgesehen, und müsste daher von Hand implementiert werden. Dahingehende Lösungsansätze sollen im kommenden Semester untersucht werden.

Kapitel 17

Datenbank-Performance

17.1 Vergleich von PostgreSQL und MongoDB

von Karl Stelzner

Als ersten Vergleich zwischen der Performance von MongoDB uns PostgreSQL haben wir die Zeit gemessen, die für die Ausführung verschiedener Selektionsanfragen auf einer Kollektion bzw. Tabelle benötigt wird. Technisch ist PostgreSQL hierbei im Vorteil, da die Einträge aufgrund des festen relationalen Datenmodells besser sequentiell von der Festplatte gelesen werden können. Die folgenden drei Queries wurden getestet:

```
1 SELECT COUNT(*) FROM events WHERE night='2013-09-29';
2 db.metaData.count({NIGHT:20130929});
3
4 SELECT COUNT(*) FROM events WHERE event_num>500 AND run_id<5;
5 db.metaData.count({EventNum:{$gt:500}, RUNID:{$lt:5}});
6
7 SELECT COUNT(*) FROM events WHERE event_num!=trigger_num;
8 db.metaData.count({$where:"this.EventNum!=this.TriggerNum"});
```

Listing 17.1: Drei Testanfragen, jeweils als SQL- und als MongoDB-Query

Das erste Query verwendet ein einfaches Gleichheitsprädikat, und kann leicht über einen Index beanwortet werden. Da die Einträge nur gezählt, und nicht ausgegeben werden, müssen die tatsächlichen Daten noch nicht einmal geladen werden. Es handelt sich somit um ein *index only query*. Ein passender Index ist auch in beiden Datenbanken vorhanden. Das zweite Query ist eine Kombination aus zwei Rangequeries, für das keine (eindimensionalen) Indizes verwendet werden können. Beide Systeme müssen daher einen vollständigen Scan über die Tabelle bzw. Collection durchführen. Die dritte Anfrage verwendet ein komplexeres Prädikat, das zwei Felder miteinander vergleicht. Für dieses Query muss das

Prädikat auf jeden Eintrag einzeln angewandt werden. Die Antwortzeiten der beiden Da-

	Query 1	Query 2	Query 3
PostgreSQL	30ms	291 ms	267ms
MongoDB	31ms	851 ms	27545 ms

Abbildung 17.1: Antwortzeit beider Datenbanken auf die oben genannten Anfragen

tenbanken auf diese Queries ist in Abbildung 17.1 dargestellt. Aus den Ergebnissen zu Query 1 lässt sich ablesen, dass beide Datenbanken indexbasierte Queries sehr schnell, und in beinahe identischer Zeit ausführen können. Man muss allerdings dazu sagen, dass PostgreSQL bei der ersten Anfrage dieser Art ca. 160ms benötigt, vermutlich, weil der Index noch in den Hauptspeicher geladen werden muss. Bei Query 2 ist PostgreSQL deutlich schneller. Dies hat vermutlich mit den oben erwähnten Unterschieden bezüglich des Datenmodells zu tun. Query 3 demonstriert, wie ineffizient MongoDB bei der Auswertung von komplexeren Anfragen ist. Während PostgreSQL hier in etwa genauso lange benötigt wie bei Query 2 (in beiden Fällen wird ein *full table scan* durchgeführt), steigt die Ausführungszeit bei MongoDB auf über 27 Sekunden an. Derartige Anfragen sind also unbedingt zu vermeiden.

Alle drei Testanfragen korrespondieren zu möglichen Anwendungen für die PG. Query 1 repräsentiert einfache Anfragen nach zum Beispiel allen Events einer bestimmten Nacht. Query 2 steht für komplexere Anfragen, die verschiedene Merkmale einschließen. Ein Beispiel wäre eine Anfrage nach allen Events, die von einer bestimmten Quelle stammen und in mondlosen Nächten aufgenommen wurden. Analyseanfragen, die verschiedene Felder miteinander in Beziehung setzen, fallen in die Kategorie von Query 3.

Da die genauen Anforderungen an die Datenbank weiterhin unklar sind, werden wir an beiden Systemen (und ElasticSearch) weiterarbeiten. Die gewonnenen Erkenntnisse werden aber für die schlussendliche Entscheidung, welche Lösung für das Endprodukt verwendet werden soll, hilfreich sein.

Kapitel 18

Fazit

von Lea Schönberger

Im Laufe des Wintersemesters 2015/2016 konnte die Projektgruppe nicht nur viele Fortschritte, sondern auch einige Erfolge verzeichnen. Über diese sei in den nachfolgenden Zeilen in resümierender Weise ein kurzer Überblick gegeben.

Zu den größten Errungenschaften dieses Semesters gehört die Realisierung verteilter Streams-Prozesse mit Apache Spark (vgl. Kapitel 15.1), mittels derer zum Einen eine Steigerung der Performanz ermöglicht wird sowie zum Anderen die Kompatibilität zum bekannten Streams-Standard gewährleistet wird. Zur Nutzung eines solchen verteilten Streams-Prozesses ist lediglich der neue Tag `distributedProcess` in der gewohnten XML-Konfiguration, der einen bereits aus Streams bekannten Multi-Stream als Input erhält, vonnöten. Doch dies ist nicht die einzige Erweiterung des Streams-Frameworks: Mithilfe des neu eingeführten `pipeline`-Tags (vgl. Kapitel 15.2.3) lässt sich darüber hinaus kinderleicht die von Spark ML bereitgestellte Pipeline via XML konfigurieren. 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` (vgl. Kapitel 15.2.2) sowie die damit verbundenen Funktionalitäten erweitert. Einen besonderen Stellenwert nimmt dabei der `operator` ein, der als neue Schnittstelle zur Bearbeitung von DataFrames (vgl. Kapitel 15.2.5) fungiert.

Doch nicht nur im gerade umrissenen Bereich wurden in diesem Semester vorzeigbare Resultate erzielt, sondern auch im Hinblick auf die Datenbankebene. Die durch das FACT-Teleskop gelieferten Meta- und Kalibrationsdaten wurden in MongoDB, Elasticsearch sowie in PostgreSQL indiziert und darüber hinaus mittels einer neu entwickelten REST-API (vgl. Kapitel 14) zugänglich gemacht. Diese soll künftig als Schnittstelle zu den (Meta- und Roh-)Daten und als Verbindungsglied zu auf diesen Daten aufbauenden Funktionalitäten dienen.

Nicht zuletzt ist auch zu erwähnen, dass die Projektgruppe in den vergangenen Monaten nicht nur auf technischer Ebene Leistungen erbracht hat, sondern auch auf strategischer

und sozialer Ebene. Während zu Beginn des Semesters die Wenigsten von uns Projekterfahrung hatten, sind uns nun Projektplanungsstrategien wie SCRUM (vgl. Kapitel 2.1.3) ein Begriff. Fernab der grauen Theorie des theoretischen Studiums konnten wir selbst Hand anlegen und im Zuge dessen unsere jeweiligen Stärken, Schwächen und Fähigkeiten, unser Zeitmanagement, unsere Belastbarkeit sowie viele weitere Aspekte ausloten und haben nun die Möglichkeit, im kommenden Semester darauf aufbauend gemeinsam unser Projekt weiterzuentwickeln und zu einem erfolgreichen Abschluss zu bringen.

Teil VI

Benutzerhandbuch

Kapitel 19

Vorbereitung eines Clusters

von Christian Pfeiffer

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 Hadoop 2.6 [7] 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 YARN [8] 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 sowie Spark 1.6 [11] zu installieren und die jeweils genannten Einrichtungsschritte zu befolgen.

Kapitel 20

Ausführung im Cluster

von Mirko Bunse

Im Cluster des Lehrstuhls läuft Spark auf YARN [8], 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.

20.1 Verfügbarkeit von Dependencies

von Mirko Bunse

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.

20.2 Komfortable Ausführung per Shell-Script

von Mirko Bunse

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 20.1 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
9  Example:
10   ./streams-submit.sh  --driver-memory 4g  example.xml
```

Listing 20.1: Verwendung des Shell-Scripts zur Ausführung im Cluster

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 20.1) 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.

Abkürzungsverzeichnis

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

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

Literaturverzeichnis

[1] *Apache Spark Resilient Distributed Datasets.* `http://www.lightbend.com/activator/template/spark-workshop`. Zugriff am 1.3.2016.

[2] *Maschinelles Lernen mit Spark MLlib.* `http://apachesparkcentral.com/category/mllib/`. Zugriff am 1.3.2016.

[3] *Lambda Architecture Illustration.* `http://data-informed.com/wp-content/uploads/2013/10/Lambda-architecture-illustration.jpg`, 2013. Zugriff am 1.3.2016.

[4] 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.

[5] Anderhub, H, M Backes, A Biland, Vittorio Boccone, I Braun, T Bretz, J Buss, Franck Cadoux, V Commichau, L Djambazov et al.: *Design and operation of FACT–the first G-APD Cherenkov telescope.* Journal of Instrumentation, 8(06):P06008, 2013.

[6] Anderhub, Hans, Adrian Biland, I. Braun, S.C. Commichau, Volker Commichau, O. Grimm, Hanspeter von Gunten, Dorothée Maria Hildebrand, Urs Horisberger, Thomas Krähenbühl, Werner Lustermann, Felicitas Pauss und et al.: *Calibrating the camera for the First G-APD Cherenkov Telescope (FACT).* Proceedings of ICRC2011, Beijing, China, 2011. ICRC.

[7] Apache Software Foundation: *Apache Hadoop*. https://hadoop.apache.org/ Letzter Zugriff: 26.03.2016.

[8] Apache Software Foundation: *Apache Hadoop NextGen MapReduce (YARN)*. https://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html 17.01.16.

[9] Apache Software Foundation: *Apache Spark Machine Learning Library Guide*. https://spark.apache.org/docs/latest/mllib-guide.html, 25.02.16.

[10] Apache Software Foundation: *Apache Storm*. http://storm.apache.org/ 17.01.16.

[11] Apache Software Foundation: *Spark - Lightning-fast cluster computing*. http://spark.apache.org/ 17.01.16.

[12] Apache Software Foundation: *Spark Streaming*. http://spark.apache.org/streaming/ 17.01.16.

[13] Atlassian: *JIRA Software*. https://de.atlassian.com/software/jira 08.02.16.

[14] Bandyopadhyay, Sanghamitra, Chris Giannella, Ujjwal Maulik, Hillol Kargupta, Kun Liu und Souptik Datta: *Clustering distributed data streams in peer-to-peer environments*. Inf. Sci., 176(14):1952–1985, 2006.

[15] Beck, Kent et al.: *Manifesto for Agile Software Development*. http://www.agilemanifesto.org/ (08.02.2016), 2001.

[16] 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*, Seiten 100–104, 2005.

[17] Bifet, Albert, Geoff Holmes, Richard Kirkby und Bernhard Pfahringer: *Moa: Massive online analysis*. The Journal of Machine Learning Research, 11:1601–1604, 2010.

[18] Birant, Derya und Alp Kut: *ST-DBSCAN: An algorithm for clustering spatial–temporal data*. Data & Knowledge Engineering, 60(1):208–221, 2007.

[19] Bock, RK, A Chilingarian, M Gaug, F Hakl, Th 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.

[20] Bockermann, Christian: *The streams Framework*. https://sfb876.de/streams/ 17.01.16.

[21] Bockermann, Christian und Hendrik Blom: *The streams Framework.* Technischer Bericht 5, TU Dortmund University, 12 2012.

[22] Bockermann, Christian, Kai Brügge, Jens Buss, Alexey Egorov, Katharina Morik, Wolfgang Rhode und Tim Ruhe: *Online Analysis of High-Volume Data Streams in Astroparticle Physics.* In: *Machine Learning and Knowledge Discovery in Databases*, Seiten 100–115. Springer, 2015.

[23] Boulicaut, Jean-Francois, Katharina Morik und Arno Siebes: *Local Pattern Detection - International Seminar Dagstuhl Castle, Germany, April 12-16, 2004, Revised Selected Papers.* Springer, Berlin, Heidelberg, 2005. Aufl. Auflage, 2005.

[24] Cappellaro, Enrico und Massimo Turatto: *Supernova types and rates.* In: *The influence of binaries on stellar population studies*, Seiten 199–214. Springer, 2001.

[25] Central, Data Sience: *Data Veracity.* `http://www.datasciencecentral.com/profiles/blogs/data-veracity`, 2012. [Online; accessed 13-March-2016].

[26] Chris McCormick: *K-Fold Cross-Validation, With MATLAB Coder.* `https:\/\/chrisjmccormick.wordpress.com\/2013\/07\/31\/k-fold-cross-validation-with-matlab-code` 29.03.16.

[27] Collaboration, FACT, T. Bretz und et al.: *Status of the First G-APD Cherenkov Telescope (FACT).* In: *Proceedings of ICRC 2011*, Beijing, China, 2011. ICRC.

[28] Corpet, Florence: *Multiple sequence alignment with hierarchical clustering.* Nucleic acids research, 16(22):10881–10890, 1988.

[29] Dean, Jeffrey und Sanjay Ghemawat: *MapReduce: Simplified Data Processing on Large Clusters.* Commun. ACM, 51(1):107–113, Januar 2008.

[30] Dietterich, Thomas G: *Ensemble methods in machine learning.* In: *Multiple classifier systems*, Seiten 1–15. Springer, 2000.

[31] Douglas, Korry und Susan Douglas: *PostgreSQL: a comprehensive guide to building, programming, and administering PostgreSQL databases.* SAMS publishing, 2003.

[32] Dries, Anton und Ulrich Rückert: *Adaptive concept drift detection.* Statistical Analysis and Data Mining, 2(5-6):311–327, 2009.

[33] Fawcett, Tom: *An introduction to ROC analysis.* Pattern recognition letters, 27(8):861–874, 2006.

[34] Fielding, Roy Thomas: *Architectural styles and the design of network-based software architectures.* Doktorarbeit, University of California, Irvine, 2000.

[35] Foundation, Apache Software: *HDFS Architecture*. `http://hadoop.apache.org/docs/r2.6.4/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html`, 2016.

[36] Freund, Yoav; Schapire, Robert E.: *A Short Introduction to Boosting*. Journal of Japanese Society for Artificial Intelligence, 14(5):771–780, 1999.

[37] Friedman, Jerome, Trevor Hastie und Robert Tibshirani: *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Springer Series in Statistics, Second Edition Auflage, 2001.

[38] Galar, Mikel, Alberto Fernandez, Edurne Barrenechea, Humberto Bustince und Francisco Herrera: *A review on ensembles for the class imbalance problem: bagging-, boosting-, and hybrid-based approaches*. Systems, Man, and Cybernetics, Part C: Applications and Reviews, IEEE Transactions on, 42(4):463–484, 2012.

[39] Gama, João, Indrė Žliobaitė, Albert Bifet, Mykola Pechenizkiy und Abdelhamid Bouchachia: *A survey on concept drift adaptation*. ACM Computing Surveys (CSUR), 46(4):44, 2014.

[40] García, Vicente, Ramón Alberto Mollineda und José Salvador Sánchez: *Index of balanced accuracy: A performance measure for skewed class distributions*. In: *Pattern Recognition and Image Analysis*, Seiten 441–448. Springer, 2009.

[41] García, Vicente, Javier Salvador Sánchez und Ramón Alberto Mollineda: *On the effectiveness of preprocessing methods when dealing with different levels of class imbalance*. Knowledge-Based Systems, 25(1):13–21, 2012.

[42] Ghemawat, Sanjay, Howard Gobioff und Shun-Tak Leung: *The Google File System*. In: *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, Seiten 29–43, New York, NY, USA, 2003. ACM.

[43] Group, FITS Working et al.: *Definition of the flexible image transport system (FITS)*. FITS Standard Version, 3, 2008.

[44] Guyon, Isabelle, Constantin Aliferis und André Elisseeff: *Causal feature selection*. Computational methods of feature selection, Seiten 63–86, 2007.

[45] Guyon, Isabelle und André Elisseeff: *An introduction to variable and feature selection*. The Journal of Machine Learning Research, 3:1157–1182, 2003.

[46] Hall, Mark A: *Correlation-based feature selection for machine learning*. Doktorarbeit, The University of Waikato, 1999.

[47] Heck, Dieter, G Schatz, J Knapp, T Thouw und JN Capdevielle: *CORSIKA: A Monte Carlo code to simulate extensive air showers*. Technischer Bericht, 1998.

[48] Helf, Marius: *Gamma-Hadron- Separation im MAGICExperiment durch verteilungsgestütztes Sampling.* Diploma Thesis, Tu Dortmund, 2011.

[49] Herrera, Franciso, Cristóbal José Carmona, Pedro González und María José del Jesus: *An overview on subgroup discovery: foundations and applications.* Knowledge and information systems, 29(3):495–525, 2011.

[50] Hortonworks: *Apache Hadoop YARN - Enabling Next Generation Data Applications.* `http://de.slideshare.net/hortonworks/apache-hadoop-yarn-enabling-nex`, 2013. [Online; accessed 23-March-2016].

[51] InterFace AG: *Das KANBAN-Plakat.* `http://www.kanban-plakat.de/` 03.11.15.

[52] InterFace AG, Technische Universität München, The Interpreneur Group: *Das SCRUM-Plakat.* `http://www.scrum-plakat.de/` 03.11.15.

[53] Jain, A. K., M. N. Murty und P. J. Flynn: *Data Clustering: A Review.* ACM Comput. Surv., 31(3):264–323, September 1999.

[54] Japkowicz, Nathalie und Shaju Stephen: *The Class Imbalance Problem: A Systematic Study.* Intell. Data Anal., 6(5):429–449, Oktober 2002.

[55] Kargupta, Hillol und Byung-Hoon Park: *A Fourier Spectrum-Based Approach to Represent Decision Trees for Mining Data Streams in Mobile Environments.* IEEE Trans. Knowl. Data Eng., 16(2):216–229, 2004.

[56] Kimball, Ralph und Margy Ross: *The data warehouse toolkit: the complete guide to dimensional modeling.* John Wiley & Sons, 2011.

[57] Kloesgen, Willi: *Explora: a multipattern and multistrategy discovery assistant.* In: Fayyad, Usama M., Gregory Piatetsky-Shapiro, Padhraic Smyth und Ramasamy Uthurusamy (Herausgeber): *Advances in Knowledge Discovery and Data Mining*, Kapitel Explora: A Multipattern and Multistrategy Discovery Assistant, Seiten 249–271. American Association for Artificial Intelligence, Menlo Park, CA, USA, 1996.

[58] Klösgen, Willi: *Applications and research problems of subgroup mining.* In: Raś, ZbigniewW. und Andrzej Skowron (Herausgeber): *Foundations of Intelligent Systems*, Band 1609 der Reihe *Lecture Notes in Computer Science*, Seiten 1–15. Springer Berlin Heidelberg, 1999.

[59] Koller, Daphne und Mehran Sahami: *Toward optimal feature selection.* 1996.

[60] Kshemkalyani, Ajay D. und Mukesh Singhal: *Distributed Computing: Principles, Algorithms, and Systems.* Cambridge University Press, Reissue Auflage, 3 2011.

[61] Lavrač, Nada, Branko Kavšek, Peter Flach und Ljupčo Todorovski: *Subgroup discovery with CN2-SD.* The Journal of Machine Learning Research, 5:153–188, 2004.

[62] Liaw, Andy; Wiener, Matthew: *Classification and Regression by RandomForest.* R News, 2, 2002.

[63] Lichman, M.: *UCI Machine Learning Repository*, 2013.

[64] Mampaey, Michael, Siegfried Nijssen, Ad Feelders und Arno Knobbe: *Efficient algorithms for finding richer subgroup descriptions in numeric and nominal data.* In: *IEEE International Conference on Data Mining*, Seiten 499–508, 2012.

[65] Marr, Bernhard: *Big Data: The 5 Vs Everyone Must Know.* https://www.linkedin.com/pulse/20140306073407-64875646-big-data-the-5-vs-everyone-must-know, 2014. [Online; accessed 13-March-2016].

[66] Marz, Nathan und James Warren: *Big Data: Principles and best practices of scalable realtime data systems.* Manning Publications, 1 Auflage, 5 2015.

[67] Marz, Nathan und James Warren: *Big Data: Principles and best practices of scalable realtime data systems.* Manning Publications Co., 2015.

[68] Masse, Mark: *REST API design rulebook.* Ö'Reilly Media, Inc.", 2011.

[69] Meier, Katja J.: *FACT - The First G-APD Cherenkov Telescope.* http://www.astro.uni-wuerzburg.de/en/research/fact/fact-introduction, Mai 2014. accessed: 23.02.2016.

[70] MongoDB, Inc.: *MongoDB.* https://www.mongodb.org/ Letzter Zugriff: 25.03.2016.

[71] MongoDB, Inc.: *MongoDB CRUD Operations.* https://docs.mongodb.org/manual/crud/ Letzter Zugriff: 25.03.2016.

[72] Morik, Katharina; Weihs, Claus: *Wissensentdeckung in Datenbank.* Folien zur gleichnamigen Vorlesung an der TU Dortmund, 2015.

[73] Nathan Marz: *A Storm is coming, Twitter Blog.* https://blog.twitter.com/2011/a-storm-is-coming-more-details-and-plans-for-release Letzter Zugriff: 25.03.2016.

[74] Nathan Marz: *Trident: a high-level abstraction for realtime computation, Twitter Blog.* https://blog.twitter.com/2012/trident-a-high-level-abstraction-for-realtime-computation Letzter Zugriff: 25.03.2016.

[75] PostgreSQL Global Development Group: *PostgreSQL*. `http://www.postgresql.org/` 16.02.16.

[76] Quinlan, J. R.:: *Bagging, Boosting, and C4.5*. In: *AAAI-96 Proceedings*, 1996.

[77] RapidMiner: *RapidMiner*. `https://rapidminer.com/` 29.02.16.

[78] Richardson, Leonard und Sam Ruby: *RESTful web services*. Ö'Reilly Media, Inc.", 2008.

[79] Rijsbergen, C. J. Van: *Information Retrieval*. Butterworth-Heinemann, Newton, MA, USA, 2nd Auflage, 1979.

[80] Saeys, Yvan, Thomas Abeel und Yves Van de Peer: *Robust feature selection using ensemble feature selection techniques*. In: *Machine learning and knowledge discovery in databases*, Seiten 313–325. Springer, 2008.

[81] Schowe, Benjamin und Katharina Morik: *Fast-ensembles of minimum redundancy feature selection*. In: *Ensembles in Machine Learning Applications*, Seiten 75–95. Springer, 2011.

[82] Settles, Burr: *Active Learning Literature Survey*. Computer Sciences Technical Report, 1648.

[83] Sharma, Mradul, Jitadeepa Nayak, Maharaj Krishna Koul, Smarajit Bose und Abhas 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.

[84] Society, The Internet: *Hypertext Transfer Protocol – HTTP/1.1*, 1999. `http://tools.ietf.org/html/rfc2616`.

[85] Society, The Internet: *The application/json Media Type for JavaScript Object Notation (JSON)*, 2006. `https://tools.ietf.org/html/rfc4627`.

[86] Todorovski, Ljupčo, Peter Flach und Nada Lavrač: *Predictive performance of weighted relative accuracy*. Springer, 2000.

[87] TU Dortmund, Fakultät für Informatik: *Modulhandbuch Master-Studiengänge Informatik und Angewandte Informatik*. `http://www.cs.tu-dortmund.de/nps/de/Studium/Ordnungen_Handbuecher_Beschluesse/Modulhandbuecher/Master_Inf/gesamtes_Modulhandbuch/Modulhandbuch_MSc_INF_1.pdf`, Dezember 2015.

[88] Turatto, Massimo: *Classification of supernovae*. In: *Supernovae and Gamma-Ray Bursters*, Seiten 21–36. Springer, 2003.

[89] Vavilapalli, Vinod Kumar, Arun C. Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, Bikas Saha, Carlo Curino, Owen O'Malley, Sanjay Radia, Benjamin Reed und Eric Baldeschwieler: *Apache Hadoop YARN: Yet Another Resource Negotiator.* In: *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, Seiten 5:1–5:16, New York, NY, USA, 2013. ACM.

[90] Wagstaff, Kiri, Claire Cardie, Seth Rogers, Stefan Schrödl et al.: *Constrained k-means clustering with background knowledge.* In: *ICML*, Band 1, Seiten 577–584, 2001.

[91] Wikipedia: *Scalability.* `https://en.wikipedia.org/wiki/Scalability`, 2016. [Online; accessed 22-February-2016].

[92] Xin, Reynold S, Josh Rosen, Matei Zaharia, Michael J Franklin, Scott Shenker und Ion Stoica: *Shark: SQL and rich analytics at scale.* In: *Proceedings of the 2013 ACM SIGMOD International Conference on Management of data*, Seiten 13–24. ACM, 2013.

[93] Zhang, Weixiong: *Complete Anytime Beam Search.* In: *Proceedings of the Fifteenth National/Tenth Conference on Artificial Intelligence/Innovative Applications of Artificial Intelligence*, AAAI '98/IAAI '98, Seiten 425–430, Menlo Park, CA, USA, 1998. American Association for Artificial Intelligence.

[94] Zhou, Zhihua: *Ensemble Methods: Foundations and Algorithms.* Chapman and Hall/CRC, 2012.