Projektgruppe 542

Stream Mining for Intrusion Detection in Distributed Systems

Endbericht

15. Oktober 2010

Matthias Balke
Tobias Beckers
Klaudia Fernys
Daniel Haak
Helge Homburg
Lukas Kalabis
Markus Kokott
Benedikt Kulmann
Kilian Müller
Carsten Przyluczky
Matthias Schulte
Marcin Skirzynski

Christian Bockermann
Benjamin Schowe

Lehrstuhl 8
Fakultät für Informatik
TU Dortmund

Wintersemester 2009/10 - Sommersemester 2010

Inhaltsverzeichnis

Vorwort

> *And so if you think about the information problem? Interesting statistics: between sorta the birth of the world and 2003, there were five exabytes of information created – that's the total over that period. In the last bit, we create 5 exabytes every two days.* (Eric Schmidt, Google CEO, 2010)

In Anbetracht der Datenmassen, die wir täglich produzieren, ist eine manuelle Auswertung und Beurteilung der Daten eine unlösbare Aufgabe. So sind statistische Lernverfahren und Data-Mining Methoden längst ein wichtiger Bestandteil der heutigen Datenanalyse geworden. Diese Verfahren zielen darauf ab, Datenbanken nach wertvollen Informationen zu durchsuchen, oder Vorhersagemodelle basierend auf gewonnenen Daten zu liefern.

Die stetig wachsende Datenflut bringt jedoch auch die verfügbaren Methoden häufig an ihre Grenzen: Gelernte Konzepte müssen kontinuierlich an neue Daten angepasst werden – sobald das Lernen eines neuen Modells länger dauert, als neue Daten anfallen, reichen bisherige Data-Mining Ansätze nicht mehr aus.

Diese neuen Anforderungen an die Datenanalyse machen einen Paradigmenwechsel erforderlich: von der Analyse statischer Datensätze zu Analyse von Datenströmen und dem *online*-Lernen von Modellen. Bei der Entwicklung von sogenannten *Stream Mining* Verfahren, wird versucht, Muster oder Modelle aus einem kontinuierlichen Strom diskreter Ereignisse (z.B. Log-Daten, Börsenkurse, medizinische Daten) abzuleiten. Eine wichtige Beschränkung ist hier, dass jedes Datum nur einmal betrachtet werden kann, d.h. die anfallenden Daten können nicht zwischengespeichert werden. Gleichzeitig spielen auch die Grenzen bezüglich des Speicherverbrauchs und der Realzeitfähigkeit der Algorithmen eine deutlich zentralere Rolle.

Ein weiterer Aspekt, der zunehmend an Bedeutung gewinnt, ist die Verteilung von Daten auf unterschiedliche Systeme. Kaum ein größeres IT-System wird heute noch in Form eines zentralisierten Systems entwickelt. Daraus folgt, dass auch die anfallenden Daten nicht immer zentral gespeichert werden (können). Zusätzlich zu den Anforderungen der Stream Mining Verfahren, ist daher auch eine Verteilung der Analysemethoden auf mehreren Systemen eine weitere wichtige Anforderung.

Beispiel: System-Überwachung

Ein Beispiel für derartige Anforderung ist die Überwachung verteilter IT-Systeme. Unter den riesigen täglich anfallenden Datenaufkommen finden sich zahlreiche unterschiedliche Log-Datensätze, die Aufschluss über die Benutzung und den Zustand von IT-Systemen geben.

Die manuelle Überwachung von verteilten IT-Systemen ist dabei aufgrund der riesigen Masse an Log-Daten nicht zu bewerkstelligen. Auch die zentrale Speicherung

der Daten verbietet sich häufig aufgrund von beschränkten Ressourcen. Hier ist eine *stream*-orientierte Analyse der Daten ein lohnenswertes Ziel. Insbesondere die Fähigkeit der Realzeit-Überwachung ist dabei ein wichtiges Merkmal um möglichst frühzeitig Angriffe oder Mißstände im System auszumachen.

PG Stream Mining for Intrusion Detection in Distributed Systems

Vor dem Hintergrund der immer weiter wachsenden Datenflut ist diese Projektgruppe entstanden. Ziel des Projektes ist die Entwicklung eines Frameworks zur Implementierung und Evaluation von Stream-Mining Verfahren innerhalb einer verteilten Umgebung. Jedes dieser Verfahren ist in Form kleiner Einheiten (Knoten) gekapselt. Innerhalb des Frameworks können diese Knoten über eine Kommunikationsstruktur miteinander interagieren, so dass ein Netzwerk intelligenter Sensoren modelliert werden kann.

Ausgehend von dem Anwendungsfall der Überwachung verteilter Web-Systeme sollen zudem auf Basis des entwickelten Frameworks Verfahren zum Monitoring verteilter Systeme entwickelt und evaluiert werden.

Der vorliegende Abschlußbericht ist das Ergebnis dieser Projektgruppe und bietet eine flexible Entwicklungsumgebung für den Aufbau eines Netzwerks von kommunizierenden Knoten. Damit lassen sich auf einfache Weise eine Vielzahl von Stream-Mining Verfahren zu einem intelligenten Sensor-Netz kombinieren und sowohl einzelne Verfahren als auch das gesamte Sensor-Netz evaluieren.

1 Einleitung

1.1 Wissenschaftlicher Kontext

Das intelligente Analysieren von Daten, hat in der Vergangenheit verschiedene Phasen durchlaufen. In jeder dieser Phasen wurden neue Erkenntnisse und Ergebnisse aus der Forschung eingebracht, um bessere Ergebnisse in kürzerer Zeit zur Verfügung zu stellen.
In der ersten Phase dieser Entwicklung wurden Daten erfasst und mit statistischen Methoden untersucht. Diese wurden im Hinblick auf eine Hypothese hin untersucht, die es zu testen galt. Mit dem Anstieg der zur Verfügung stehenden Rechenleistungen gewann auch das Gebiet des Maschinellen Lernens immer mehr an Bedeutung. Durch die Entwicklungen, welche auf diesem Gebiet im wissenschaftlichen Bereich erfolgten, konnten neue Analyseprobleme adressiert werden. Probleme entstanden vor allem durch die ständig anwachsenden Datenbankgrößen und so mussten neue, bessere Domänenspezifisch Algorithmen entworfen werden. Diese sollten besser skalierbar sein, um auch für zukünftige Einsätze, bei noch größeren Datenbeständen, zur Verfügung zu stehen. Mit der Zeit wurden immer mehr Verfahren aus dem maschinellen Lernen und der statistischen Analyse zusammengefügt um Modelle und Muster von sehr großen Datenbankbeständen heraus zu gewinnen.
Fortschritte bei Netzwerktechniken, speziell beim parallelen Rechnen führten dazu, dass die Parallelisierung auch beim *data mining* mehr und mehr Einzug gehalten hat. Ziel des Ganzen war herauszufinden, wie man Wissen aus verschiedenen Teilmengen extrahiert und dieses neu gewonnene Wissen in strukturierter Art und Weise in ein Globales Modell integriert.
Um die Jahrtausendwende herum, wuchs schließlich die Zahl der Datengenerierung so rapide an, dass erneut neue Verfahren gebraucht wurden, um mit diesen neu hinzukommenden Datenmengen umgehen zu können.

Mit genau diesen Verfahren beschäftigt sich diese Projektgruppe. Wie bereits erwähnt, basierten früherer Methoden darauf, Daten zu sammeln, diese auf einem Datenträger zu speichern um sie dann später mit analytischen Methoden zu untersuchen und auszuwerten. Sowohl der Speicherplatz, wie auch die Zeit sind dabei zwei Komponenten, die es zu vernachlässigen gilt. Im *data stream* Bereich sieht die Situation jedoch anders aus. Hier muss man sich zum Einen der Aufgabe stellen, dass der zur Verfügung gestellte Speicherplatz relativ gering ist und zum Anderen, dass die zeitliche Komponente eine wesentliche Rolle spielt.
Systeme die diese Art von Daten schnell und zuverlässig verarbeiten müssen sind z.B. die sog. Intrusion-Detection-Systeme. Diese haben die Aufgabe, Netzwerkverkehr zu analysieren, nach bestimmten Mustern oder Anomalien Ausschau zu halten und gegebenenfalls darauf zu reagieren. Wegen den bereits erwähnten Restriktionen die es einzuhalten gilt, können die zu analysierenden Daten häufig nur ein einziges Mal betrachtet werden, bevor sie wieder verworfen werden und das nächsten Daten-

paket bearbeitet wird. Ziel muss es also sein, effiziente Methoden und Verfahren zu entwickeln, die in der Lage sind, die geforderten Anforderungen zu leisten.
Diese neuen Anforderungen haben dieses Gebiet zu einem aktuellen Forschungsthema werden lassen. So gibt es, neben einer Vielzahl nationaler Konferenzen, gerade im internationalen Bereich einige Top-Konferenzen, die sich mit diesem Thema befassen (z.B. ICDM, KDD, SDM etc.).

1.2 Ziele

Ziel der Projektgruppe *Stream Mining for Intrusion Detection in Distributed Systems* ist es, ein Framework zur Stream-Mining Analyse zu entwickeln. Zur Klärung der Einzelziele werden, zusammen mit den Veranstaltern der Projektgruppe, ein Pflichtenheft erarbeitet. In diesem werden die Anforderungen, die die Veranstalter an die Ergebnisse der Gruppe haben, festgehalten. Das folgende Kapitel stellt einen Auszug aus dem erarbeiteten Pflichtenheft dar.

1.2.1 Minimalziele

Dieser Abschnitt beschreibt die Anforderungen, die von der Projektgruppe zu erfüllen sind. Die Anforderungen sind in folgende Abschnitte gegliedert:

1. Vorphase
2. Architektur und Design
3. Implementierung
4. Evaluierung
5. Dokumentation

Die Reihenfolge der Bearbeitung ist, abgesehen von möglichen Abhängigkeiten der Aspekte, frei. Jeder dieser Aspekte ist zu dokumentieren.

Zum Ende des ersten Semesters soll eine grundlegende Version des Frameworks vorliegen. Des Weiteren sollen zwei Anwendungsfälle realisiert werden. Bei den Anwendungsfällen hat sich die Projektgruppe auf die Analyse von Logfiles und die Spam-Erkennung geeinigt.

Während des zweiten Semesters soll das Framework weiter entwickelt werden. Des Weiteren soll das Framework um Stream-Mining Algorithmen erweitert werden. Die Evaluierung verschiedener Modelle soll gewährleistet werden. Wie im ersten Semester soll auch im zweiten Semester ein Anwendungsfall realisiert werden. Hierbei hat sich die Projektgruppe für die Web-Anomalie-Erkennung entschieden.

Vorphase Innerhalb der Projektgruppe soll das in der Einleitung beschriebene Framework entworfen und implementiert werden. Zum Entwurf sollen dazu verschiedene Fallbeispiele modelliert und mit Hilfe des Frameworks realisiert werden können. Zu diesem Zweck sollen als Grundlage für den Architekturentwurf zunächst verschiedene Anwendungsfälle aufgestellt und beschrieben werden. Auf Basis der während der Seminarphase gehaltenen Vorträge zum Thema IT Sicherheit, sollen dazu für das erste Halbjahr zwei Fallbeispiele festgelegt werden. Mögliche Domänen für derartige Beispiele wären z.B. die Analyse verteilter Log-Daten von Unix-Systemen, die Analyse von Netzwerkdaten innerhalb eines lokalen Netzes oder die Erkennung von Spam Emails. Insgesamt sollen innerhalb der Vorphase also folgende Anforderungen erfüllt werden:

1. Entwicklung von zwei Anwendungsfällen, die innerhalb des ersten Semesters als Grundlage für die Modellierung und Implementierung dienen.
2. Anforderungsanalyse an das Framework.

Architektur und Design Basierend auf den erarbeiteten Anforderungen und Fallbeispielen, soll eine Architektur entwickelt werden. Der Architekturentwurf soll dann in Form eines Feinkonzeptes dargestellt und beschrieben werden. Dieses Feinkonzept bildet zugleich die Grundlage für die Implementierung des Frameworks. Neben den in der Vorphase herausgearbeiteten Anforderungen, soll die Architektur des Netzes folgende Eigenschaften erfüllen:

1. **Festlegung der Netzstruktur durch den Benutzer**
   Es muss dem Nutzer des Frameworks möglich sein, die Struktur des Frameworks selbstständig festzulegen. Hierzu müssen die einzelnen Abhängigkeiten festgelegt werden.
2. **Bereitstellung von Lernknoten**
   Das Framework soll bereits eine Menge von Knoten mit implementierten Lern-Algorithmen und Eingabeknoten bereitstellen, mit denen die Fallbeispiele aus der Vorphase umgesetzt werden können. Eine Anforderung ist, dass jeder PG-Teilnehmer mindestens eine Implementierung eines Stream-Algorithmus mit Hilfe der API des Frameworks umsetzt.
3. **Möglichkeit der Erweiterung**
   Das Framework soll über die Bereitstellung einer (einfachen) API in Form von Interface-Definitionen um weitere Stream-Algorithmen oder reaktive Komponenten erweitert werden können.
4. **Evaluierung von Algorithmenknoten**
   Neben dem Einsatz von einfachen, stream-orientierten Lernverfahren, soll das Framework die Evaluierung von verschiedenen Lernverfahren ermöglichen. Zur Evaluierung sind Vergleiche der erlernten Modelle oder die Bewertung gelabelter Daten vorstellbar. Dazu sollen einfache Strukturen bereitgestellt werden, die

die Evaluierung von Stream-Algorithmen und den Vergleich dieser Algorithmen mit anderen Stream-Algorithmen oder Batch-Lernverfahren unterstützen. Die Umsetzung der Batch-Lernverfahren ist nicht Aufgabe der Projektgruppe. Insbesondere sollen diese Evaluierungsmöglichkeiten in der zweiten Projektphase genutzt werden, um die implementierten Algorithmen für die gewählten Fallbeispiele zu vergleichen.

Implementierung Die entworfene Architektur soll in einer realen Implementierung umgesetzt werden. Die verwendete Programmiersprache ist frei wählbar. Die Implementierung soll die während der Vorphase herausgearbeiteten Anforderungen, sowie die im Architektur und Design Kapitel beschriebenen Anforderungen erfüllen. Zudem sollen ≤ 12 Stream-Mining Verfahren aus den Bereichen:

- Zählen, häufige Mengen
- Klassifikation
- Regression
- Unüberwachtes Lernen

realisiert werden.

Evaluation Wie im Kapitel Architektur und Design unter dem Punkt Evaluierung von Algorithmenknoten bereits erwähnt, sollen die implementierten Stream-Algorithmen hinsichtlich verschiedener Kriterien evaluiert werden, sofern diese auf den Algorithmus sinnvoll anwendbar sind. Dies soll auf mehreren gelabelten Datensätzen durchgeführt werden.

Zum anderen sollen die Algorithmen untereinander und mit Batch-Algorithmen verglichen werden, in Bezug auf:

1. Güte des Modells
2. Geschwindigkeit
3. Speicherverbrauch
4. Benötigte Kommunikation
5. Effizienz

Dokumentation Die Arbeit der Projektgruppe muss in schriftlicher Form dokumentiert werden. Die Dokumentation soll sowohl die Architekturbeschreibung, die Anwendungsfälle und die Implementierung umfassen. Weiterhin ist für die Nutzung der Implementierung eine Dokumentation der API erforderlich. Zusammenfassend soll die Dokumentation aus folgenden Teilen bestehen:

1. Beschreibung von Fallbeispielen
2. Ausführliche Beschreibung der Ergebnisse der Anforderungsanalyse
3. Darstellung der entwickelten Framework-Architektur
4. Dokumentation der Programmierschnittstellen (API)
5. Handbuch zur Benutzung des Frameworks
6. Beschreibung und Evaluierung der implementierten Stream-Algorithmen
7. Abschlusspräsentation

Die Dokumentation sollte kontinuierlich während der Arbeitsphasen (Semester) gepflegt werden. Zum Ende des ersten Semesters ist ein Zwischenbericht abzuliefern, der die Arbeit des ersten Semesters dokumentiert und zusätzlich eine Übersicht über weitere Schritte im Folgesemester enthält.

Das Ergebnis der gesamten Projektgruppe (1. & 2. Semester) wird in Form eines umfangreichen Endberichts dokumentiert. Dieser Endbericht wird zudem als Forschungsarbeit der gesamten Arbeitsgruppe veröffentlicht (als interner Bericht der TU Dortmund).

1.2.2 Erweiterungen

Über die in den Minimalzielen festgehaltenen Anforderungen hinaus, hat das Projektteam weitere Anforderungen identifiziert. Diese sollen, soweit möglich, ebenfalls umgesetzt werden. Zu diesen Anforderungen zählen:

- Validierung und Testen der Implementierung
- Implementierung reaktiver Komponenten

Im Folgenden werden diese möglichen Erweiterungen genauer spezifiziert.

Validierung und Testen Es wäre wünschenswert, für relevante Komponenten des Frameworks Test-Methoden (z.B. in Form von Unit-Tests) zu entwickeln. Die Verwendung von Tests ist keine Pflicht, jedoch sollte eine Validierung der implementierten Algorithmen auf Richtigkeit ermöglicht werden.

Implementierung reaktiver Komponenten Vorstellbar ist, dass das Framework nicht nur auf den Streams lernt, sondern die erlernten Ergebnisse sofort in eigenständige Reaktionen umsetzt. Eine solche reaktive Komponente könnte in einigen Anwendungsbereichen (z.B. Intrusion Detection) sinnvoll sein.

1.3 Teammitglieder

Von Seiten des Lehrstuhls 8 wird das Projekt betreut durch

- Dipl.-Inf. Christian Bockermann und
- Dipl.-Inf. Benjamin Schowe.

Teilnehmer an der Projektgruppe sind

- Balke, Matthias
- Beckers, Tobias
- Fernys, Klaudia
- Haak, Daniel
- Homburg, Helge
- Kalabis, Lukas
- Kokott, Markus
- Kulmann, Benedikt
- Müller, Kilian
- Przyluczky, Carsten
- Schulte, Matthias
- Skirzynski, Marcin.

Die Teilnehmer der Projektgruppe sind gemeinschaftlich Autoren dieses Berichtes.

1.4 Aufbau des Dokuments

Nachdem wir im diesem Kapitel bereits die Ausgangslage sowie die Aufgabenstellung für unsere Projektgruppe näher erläutert haben, gehen wir in den folgenden Kapitel nun auf die von uns erarbeiteten Ergebnissen ein.

Im Kapitel 2 gehen wir zunächst auf organisatorische Konventionen ein, die wir festgelegt haben um die Grundlagen für ein strukturiertes Arbeiten im Team zu schaffen. Hierzu zählt vor allem

- die Festlegung von festen Terminen für Gruppentreffen,
- die Aufgabenverteilung im Team,
- die Festlegung auf ein Vorgehensmodell sowie
- die Abstimmung eines verbindlichen Zeitplans,
- die genaue Definition strittiger Begriffe,
- die begründbare Entscheidung für eine Programmiersprache und
- die Auswahl zu verwendender Tools.

Im Kapitel 3 wenden wir uns den fachlichen Grundlagen zu, die wir für die beiden mehrtägigen Seminarfahrt ins Gästehaus der Technischen Universität Dortmund nach Bommerholz am Anfang des Wintersemesters, sowie zu Beginn des Sommersemesters erarbeitet haben. Das Kapitel eignet sich hervorragend um einen Überblick über die für unsere Projektgruppe wichtigen Teilbereiche der Fachgebiete „Data Mining“, „Stream Mining“ und „maschinelles Lernen“ zu bekommen. Die in diesem Kapitel angesprochenen Themen stellen die Grundlage für die Entwicklung unseres Frameworks dar.

Im Folgenden Kapitel 4 gehen wir dann näher auf das von uns entwickelte Framework ein. Die Grundideen des Zusammenspiels der verschiedenen Komponenten werden hier erläutert und es wird ein Überblick über die technische Umsetzung der Kernfunktionalitäten unseres Projekts gegeben. Das Kapitel richtet sich grundsätzlich an jeden Leser, es sei aber insbesondere den - hoffentlich zahlreichen - Nutzern unseres Frameworks ans Herz gelegt.

Neben der einfachen und effizienten Anwendung der implementierten generischen Lernverfahren, die in Kapitel 6 näher erläutert werden, war der Projektgruppe vor Allem die Bereitstellung einer intuitiven Evaluationsumgebung für die Verfahren wichtig. Diese Evaluationsumgebung wird in Kapitel 5 detailliert beschrieben.

Im letzten Kapitel widmen wir uns dann dem Anwendungsfall *Web Anomalie Detection System*. Basierend auf einem Paper von Kruegel, Vigna und Robertson [30]

haben wir unser Framework dazu verwendet, ein Erkennungssystem für webbasierte Angriffe in verteilten Rechnernetzen zu entwickeln. Es hat sich gezeigt, dass unser Framework für eine solche Anwendung sehr gut geeignet ist.

Auf eine Dokumentation des Anwendungsfalles „Spamerkennung“ und „Intrusion Detection auf Basis von SSH Logfiles“, die wir zu Beginn der Projektgruppe im Wintersemester realisiert haben, haben wir im Rahmen dieses Berichtes verzichtet. Dem interessierten Leser sei hier der Zwischenbericht [3] unserer Projektgruppe nahegelegt.

2 Projektkonventionen

2.1 Gruppentreffen

Alle Projektgruppenmitglieder treffen sich regelmäßig im großen Plenum. Die Treffen dauern in der Regel zwischen zwei und vier Stunden. Sie dienen vor Allem der Vorstellung der Ergebnisse, die einzelne Arbeitsgruppen erarbeitet haben. Des Weiteren wird bei den Treffen die Aufgabenverteilung durchgeführt. Die konkreten Aufgaben werden außerhalb der Gruppentreffen von kleineren Arbeitsgruppen erarbeitet.
Die Treffen im großen Plenum sind von den gewählten Projektleitern moderiert. Von jedem Treffen wird ein Protokoll angefertigt, welches die gefassten Beschlüsse dokumentiert. Auf Basis des Protokolls und der Zeitplanung erstellen die Projektleiter dann eine Tagesordnung für das nächste Treffen. Feste Bestandteile dieser Tagesordnung sind

- die Bestimmung eines Protokollanten,
- die Genehmigung des letzten Protokolls,
- die Vorstellung der Ergebnisse aus den Arbeitsgruppen sowie
- die Planung der Aufgaben bis zur nächsten Sitzung.

Die Treffen der Arbeitsgruppen werden nach Bedarf und zeitlicher Verfügbarkeit der Arbeitsgruppenmitglieder selbstständig terminiert und organisiert.

2.2 Aufgaben der Projektleiter

Die Projektgruppe ist für viele Studenten das erste größere Softwareprojekt an dem sie als vollwertiges Teammitglied teilnehmen. Die Fähigkeiten und Erfahrungen der Studierenden sind verständlicherweise auf unterschiedlichem Niveau, die Stärken und Schwächen auf viele Themengebiete verteilt. Dies kann schnell zu Uneinigkeiten und langen, hitzigen Diskussionen führen, gerade in einem recht großen Team. Neben den fachlichen gilt es daher auch soziale Kompetenzen (weiter) zu entwickelt, um die Projektgruppe zu einem erfolgreichen Ergebnis zu führen. Selbstorganisation durch ein umfangreiches Projektmanagement ist damit unerlässlich und gehört mit zu den wichtigsten Aufgaben des Projekts.
Während in einem Softwareprojekt in der Wirtschaft die Rollen und Kompetenzen der Teammitglieder meist von Anfang an klar verteilt sind, handelt es sich bei den Mitgliedern der Projektgruppe um eine homogene Gruppe von Studierenden. Daher stellte sich uns zunächst die Frage, ob wir für unsere Projektgruppe die Rolle des Projektleiters überhaupt besetzen wollen. Letztendlich haben wir uns dazu entschieden, die Rolle als moderierende, koordinierende Instanz zu nutzen, welche die Überwachung des Projektmanagement als Hauptaufgabe hat, den anderen Teammitgliedern

in allen anderen Belangen aber vollkommen gleichgestellt ist. Vor allem für die Organisation und Koordination der Gruppentreffen hat sich die Einführung der Rolle als richtige Entscheidung erwiesen. Als Projektleiter wurden Matthias Balke und Kilian Müller gewählt.

Die Aufgaben der Projektleitung liegen wie bereits erwähnt in der Vorbereitung der Teammeetings, der moderierenden Leitung der einzelnen Sitzungen, der Überwachung der Projektziele und der Koordinierung der einzelnen Arbeitsschritte. Ebenso gehören zu ihren Aufgaben die Erstellung und Verwaltung des Terminplans. Während der Sitzungen achten sie auf die Einhaltung der Tagesordnung und der vorgesehenen Zeiteinteilung. Wenn sich eine Diskussion im Kreis dreht oder für den Rahmen des Gruppentreffens zu weit führt, weisen sie darauf hin und machen Vorschläge wie die Probleme gelöst werden können.

2.3 Verfahrensmodell

Dieses Kapitel erläutert die einzelnen Verfahrensmodelle, über die wir uns am Anfang der Projektgruppe informiert haben.

2.3.1 Wasserfallmodell

Das herkömmliche Wasserfallmodell beinhaltet die folgenden Phasen:

1. Initialisierung
2. Analyse
3. Entwurf
4. Realisierung
5. Einführung
6. Nutzung

Dabei ist es lediglich möglich von einer oberen Phase in die direkt darunter liegende Phase zu gelangen, siehe Abbildung 1. Dies ist problematisch, da Fehler aus der Analyse-Phase, die erst bei der Realisierung bekannt werden, nicht direkt behoben werden können. Es müssten alle Phasen bis zur Nutzung durchlaufen werden. Anschließend muss das Wasserfallmodell von Beginn an neu durchlaufen werden. Bei vielen Änderungen ist dies sehr mühsam und kostspielig.

Daher wurde in Laufe der Zeit das Modell durch einen Rückfluss erweitert, siehe Abbildung 1. Hierbei kann man bei Veränderungswünschen, die man erst in nachfolgenden Phasen feststellt auch direkt zurückgehen ohne das gesamte Modell bis zum

Ende durchlaufen zu müssen. Daher müssen lediglich die Zwischenphasen zusätzlich durchlaufen werden.

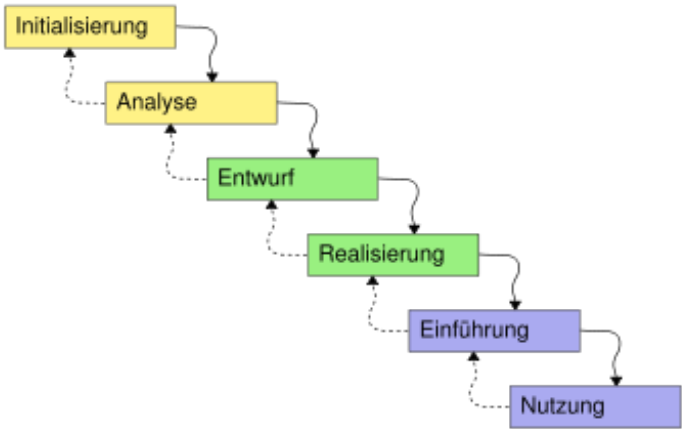


Abbildung 1: *Verfahrensmodell: Wasserfallmodell [17]*

2.3.2 V-Modell

Das V-Modell ist ebenfalls eine Weiterentwicklung des Wasserfallmodells. Dabei sind nun die einzelnen Tests, die für die Kontrolle der einzelnen Phasen benötigt werden, hinzugekommen. Das V-Modell hat auch einen gradlinigen Verlauf, sprich es ist nicht möglich aus der Design-Phase ohne weiteres wieder zurück in die Analyse zu gelangen. Werden Fehler eines Vorgängers erst in der nächsten Phase erkannt so muss weiter nach dem Modell vorgegangen werden. Erst nach dem Testen der Phase ist es möglich wieder zurück in die Phase selbst zu gelangen. Dies ist ebenfalls kostspielig.

2.3.3 Scrum Modell

Die Haupteigenschaften eines agilen Modells sind eine hohe Eigenverantwortung der Gruppenmitglieder und die Möglichkeit schnell auf Änderungen einzugehen.

Am Scrum-Model sind der *Scrum-Master*, der *Product-Owner* und das Team beteiligt. Hierbei ersetzt der Scrum-Master die Rolle des herkömmlichen Projektleiters. Das Zuweisen der Aufgaben erfolgt nicht mehr über den Scrum-Master. Die Gruppenmitglieder wählen ihre Aufgaben selbstständig. Eine Aufgabe des Scrum-Masters ist, sich um den Verlauf und das Einhalten von *Meetings* zu bemühen.

Um die inhaltlichen Punkte kümmert sich der Produkt-Owner. Dieser geht mit dem Aufgabensteller (Kunden) die Anforderungen durch und erstellt ein *Product Backlog*. Nach der Erstellung kommt es zur weiteren Analyse der einzelnen Anforderungen.

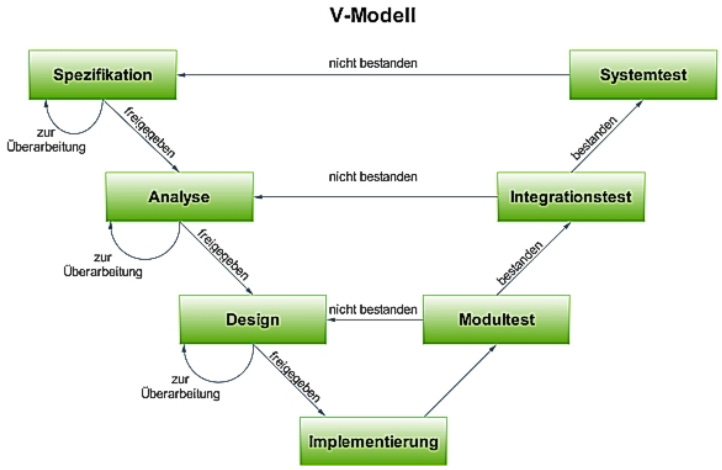


Abbildung 2: *Verfahrensmodell: V- Modell [7]*

Diese werden nun mit Prioritäten und kurzen Erläuterungen versehen. Anschließend setzt sich ein Team aus ca. drei Mitarbeitern zusammen und schätzt den jeweiligen Aufwand für die einzelnen Anforderungen. So können die Anforderungen, die im Product Backlog stehen, im nächsten Schritt aufgeteilt werden. Diese werden als *Sprint Backlogs* bezeichnet und haben eine Bearbeitungszeit von zwei bis vier Wochen. Nach jedem Abschluss eines Sprint Backlogs sollte ein Feature des Programms fertig gestellt sein. Die einzelnen Sprint Backlogs bauen aufeinander auf. Zudem sind diese nach Prioritäten der Funktionen sortiert. Nun können die einzelnen Teammitglieder die Aufgaben auswählen.

Das Scrum-Modell sieht täglich ein kurzes sogenanntes *Daily Scrum Meeting* vor. Bei diesem Meeting beschreibt jeder Mitarbeiter kurz seinen jetzigen Stand, was er bis zum nächsten Meeting erreichen möchte und falls er Schwierigkeiten hat, woran diese liegen. Zur Besprechung der einzelnen Probleme kommt es jedoch erst nach dem Daily Scrum Meeting. Dabei setzen sich die sogenannten *Team-Leads* mit den einzelnen Personen zusammen und besprechen das weitere Vorgehen. Zur Präsentation der einzelnen Features kommt es erst nach einem abgeschlossenen Sprint Backlog. Dabei präsentiert jeder seine eigene Arbeit. Dies hat den Vorteil, dass jeder die Meinung vom Kunden direkt erhält.

2.3.4 Inkrementelles Prototyping

Unter inkrementellem Prototyping versteht man das schrittweise Bearbeiten eines Projektes, bei dem zu Beginn noch nicht das endgültige Ziel feststeht. Dabei werden die einzelnen Prozesse einzeln entwickelt und kontinuierlich verbessert. Nach dem Abschließen einer Entwicklungsphase werden die Teilsysteme in das Projekt integriert. Ein großer Vorteil dabei ist, dass schnellst möglich mit einem Prototyp begonnen wird und so die ersten Probleme und Veränderungen sichtbar werden.

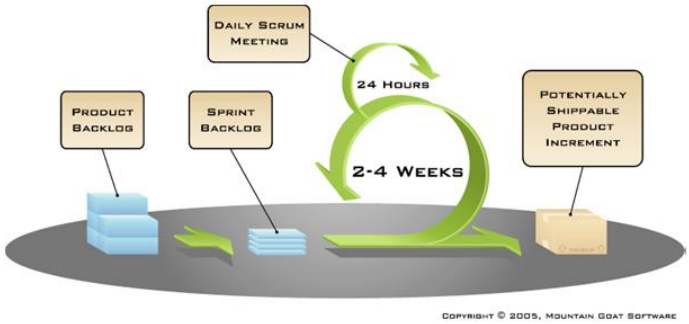


Abbildung 3: *Verfahrensmodell: Scrum Modell [19]*

Nachdem ein Teilsystem fertig gestellt wird muss dieses nicht mehr neue betrachtet werden, wenn Veränderungen bei anderen Teilen vorgenommen werden. Das Ergebnis einer Iteration hingegen wird auf notwendige Änderungen untersucht, vor allem hinsichtlich einer Anpassung der Ziele späterer Iterationen. Der Hintergrund dieses Modells liegt darin, dass die Entwickler eine bessere Möglichkeit erhalten, die Erfahrungen aus den abgeschlossenen Phasen mit in die neuen Phasen aufzunehmen, somit gestaltet sich Weiterentwicklung der einzelnen Teilsysteme leichter.

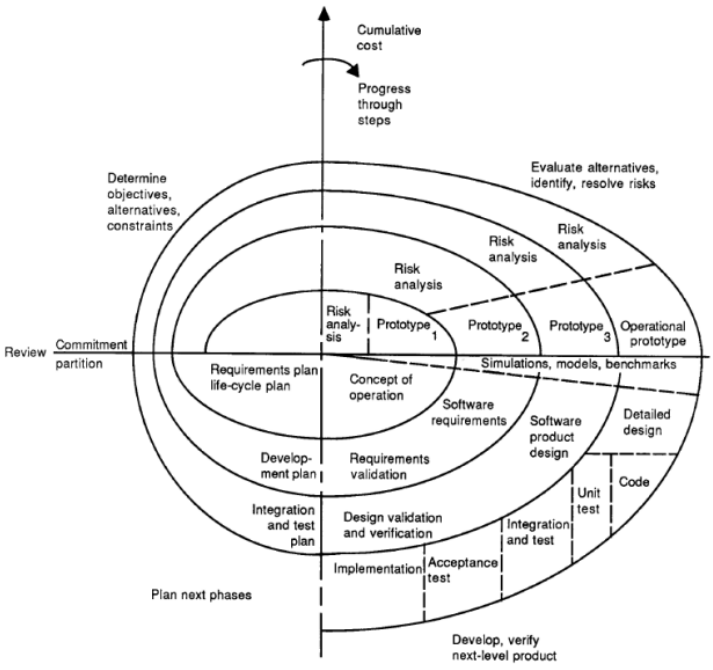


Abbildung 4: *Verfahrensmodell: Inkrementelles Prototyping [39]*

2.3.5 Fazit

Für diese Projektgruppe haben wir uns für das Inkrementelle Prototyping entschieden. Da hierbei schon zu Beginn auf einem Prototyp aufgebaut wird, welcher dann im Laufe der Zeit kontinuierlich verbessert und an die Gegebenheiten angepasst wird. Dies hat den Vorteil, dass es nicht so zeitaufwendig wie das Wasserfallmodell und das V-Modell ist. Jedoch bietet auch das Scrum-Modell Vorteile für unser Projekt, die wir mit einfließen lassen. Diese sind, die Treffen, die bei uns zweimal wöchentlich stattfinden. Bei diesen Treffen präsentiert jede Gruppe kurz den Stand der Entwicklung und erläutert eventuelle Schwierigkeiten. Die Bildung von kleineren Teams hat den Vorteil, dass so parallel an mehreren Sachen gearbeitet wird.

2.4 Zeitplan

Nachdem im ersten Halbjahr der Projektlaufzeit bereits die Grundpfeiler des Frameworks entwickelt wurden, geht es im zweiten Halbjahr vor allem um die Entwicklung von Algorithmen. Neben verschiedenen generischen Algorithmen, die für spätere Arbeiten mit dem Framework bereitgestellt werden sollen, geht es aber auch um die Entwicklung eines weiteren Anwendungsfalls. Selbstverständlich mit einer Realisierung auf Grundlage der geschaffenen Umgebung.
Generell basiert der Zeitplan des Sommersemesters 2010/2011 auf größerem Freiraum für die Zeiteinteilung der Teilnehmer. Daher wurden nur grobe Zeitplanungen vorgenommen und vereinzelte Meilensteine festgelegt:

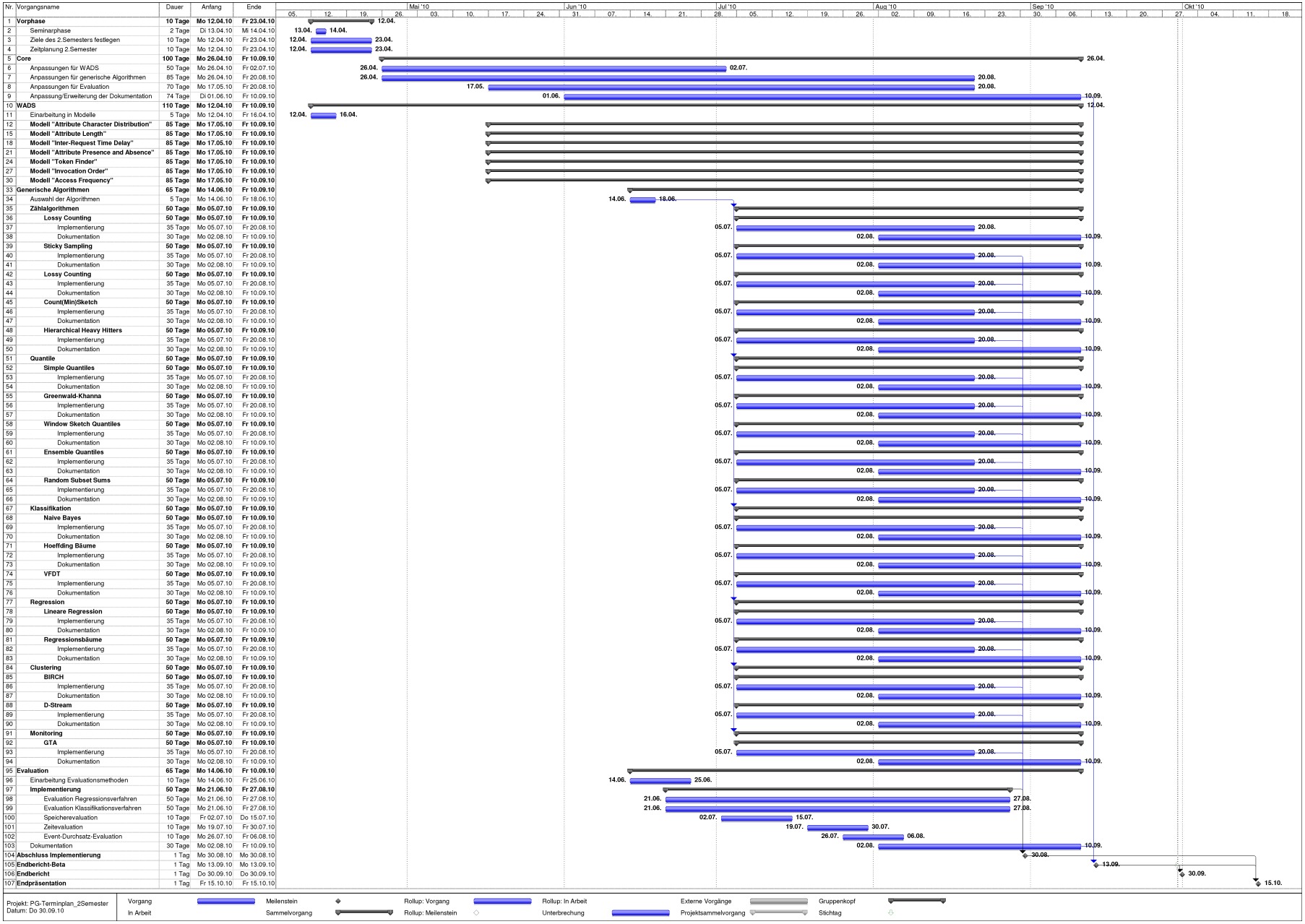


Abbildung 5: *Terminplan für das Sommersemester 2010/2011*

2.5 Programmiersprache

In diesem Kapitel geht es um die Kriterien zur Auswahl der Programmiersprache für das Projekt.

Anhand der Ziele der Projektgruppe lassen sich einige Anforderungen an die Programmiersprache ausmachen.
Da unser Netzwerk auf verschiedene Computer verteilbar sein soll, ist es wichtig, dass die gewählte Programmiersprache einfache Möglichkeiten bietet, eine Kommunikation unter den einzelnen Containern zu ermöglichen. Des Weiteren ist es wünschenswert, eine vom Betriebssystem unabhängige Programmiersprache zu wählen, da dadurch eine größere Auswahl an Computern als Grundlage für unsere Container zur Verfügung stehen. Zusätzlich sollte es sich um eine objektorientierte Sprache handeln. Wünschenswert wäre es eine Sprache zu benutzen, die leicht für Neueinsteiger zu erlernen ist, damit sich die Weiterentwicklung so einfach wie möglich gestaltet.

Die von uns gestellten Anforderungen werden alle von der Programmiersprache Java erfüllt. Java wurde entwickelt, um verteiltes Programmieren zu gewährleisten. Es ist trotzdem eine relativ einfach zu erlernende Sprache, die von Haus aus schon über viele Funktionalitäten und Datenstrukturen verfügt. Sie ist sehr gut dokumentiert und weit verbreitet, wodurch sie sich hervorragend für unser Projekt eignet. Da es zugleich die am meisten verwendete Sprache in unserem Studiengang ist, kennen sich alle Projektgruppenteilnehmer mit dieser bereits sehr gut aus. Dies verringert den anfänglichen Umstellungsaufwand enorm. Auch bezüglich Erweiterbarkeit ist Java eine gute Wahl, da es z.B. auch Schnittstellen zu vielen Datenbanken bietet oder die Anbindung an *Web Services* unterstützt.

2.6 Verwendete Tools

2.6.1 Eclipse

In jedem größeren Softwareprojekt sind leistungsstarke Entwicklungsumgebungen unverzichtbar geworden. Zwar wäre die Entwicklung auch mit klassischen Editoren, wie *vim*, *emacs*, *notepad* etc. theoretisch möglich, jedoch leidet die Übersichtlichkeit und der Wartungsaufwand ab einer bestimmten Projektgröße erheblich darunter. Auf dem Markt ist eine Vielzahl von Entwicklungsumgebungen sowohl kommerzieller Natur wie auch als freie Lösung erhältlich.
Wir haben uns in unserer Projektgruppe für die *open source* Lösung *Eclipse* entschieden, welche im Folgenden kurz beschrieben wird.
Ursprünglich wurde Eclipse als reine Entwicklungsumgebung für die Sprache Java

konzipiert. Jedoch kann das Grundgerüst, welches selbst komplett in Java geschrieben ist, durch eine Vielzahl von *plugins* an die eigenen Bedürfnisse angepasst werden. Dadurch ist man nicht mehr auf eine einzige Programmiersprache beschränkt. Da die grafische Oberfläche mit dem ***s**tandard **w**idget **t**oolkit (SWT)* implementiert wurde, welches genauso wie *AWK* auf den naiven GUI-Komponenten des Betriebssystems aufsetzt, kann die von Java gewohnte Plattformunabhängigkeit leider nicht gewährleistet werden. Jedoch existieren für Eclipse 14 verschiedene Implementierungen, so dass die Nutzung unter nahezu jedem Betriebsystem und allen gängigen Architekturen möglich ist. Aktuell liegt Eclipse in der Version 3.5 (Projektname: Galileo) vor.
Neben der freien Verfügbarkeit und großer Akzeptanz dieses Werkzeugs in der Gemeinschaft ist die durch plugins gewährleistete Erweiterbarkeit Grund für den Einsatz von Eclipse in unserem Projekt. So ist es uns möglich neben der reinen Entwicklungsumgebung auch andere, für die Softwareentwicklung unverzichtbare Programme zu nutzen, bzw. direkt in Eclipse zu integrieren. Dieses sind zum Beispiel Versionierungstools - in unserem Fall Subversion (siehe 2.6.3) - oder auch *build management tools* (siehe 2.6.5). Beides kann in Eclipse durch Plugins integriert werden, so dass all diese Werkzeuge in einer Umgebung zusammenarbeiten können.

2.6.2 Latex

LaTeX ist ein Softwarepaket, das die Benutzung des Textsatzprogramms TeX mit Hilfe von Makros vereinfacht. Die aktuelle Version ist 2ε. LaTeX ist frei verfügbar und unabhängig von der benutzen Hardware und dem Betriebssystem. Im Gegensatz zu einem gewöhnlichen Textverarbeitungsprogramm, wie z.B. Word, ist LaTeX kein Editor nach dem *WYSIWYG-Prinzip („**w**hat **y**ou **s**ee **i**s **w**hat **y**ou **g**et")*. Somit sieht der Autor beim Verfassen des Textes nicht das endgültige Format des Dokuments. Spezielle Formatierungen, wie z.B. Überschriften, Absätze etc., können durch Befehle gekennzeichnet werden. Das Layout von LaTeX gilt dabei als sehr sauber. Auch ist eine einfache Portierung in die Formate PostScript, HTML oder PDF möglich. LaTeX eignet sich insbesondere für umfangreiche wissenschaftliche Arbeiten und Berichte. Dadurch dass Dokumente eingebunden werden können, ist es möglich, verteilt mit mehreren Benutzern gleichzeitig an größeren Dokumenten zu arbeiten. Durch die komfortablen Möglichkeiten der Formelsetzung hat sich LaTeX vor allem in der Mathematik und im naturwissenschaftlichen Bereich durchgesetzt. Es gibt eine Menge an Zusatzprogrammen für LaTeX, die diverse Funktionen zu einem Paket zusammenfassen. Prinzipiell ist es möglich, TeX Dokumente mit einem simplen Editor zu erstellen. Zum Einsatz kommt LaTeX in unserem Fall bei dem Zwischen- bzw. Endbericht sowie für die Sitzungsprotokolle.

2.6.3 SVN

Subversion (SVN) ist ein frei verfügbares Versionierungssystem für Dateien und Verzeichnisse. Änderungen, welche im Laufe der Zeit durch verschiedene Entwickler bei einem Softwareprojekt entstehen, lassen sich somit auf komfortable Art und Weise verwalten. Des Weiteren stellt SVN den Benutzern eine komfortable Möglichkeit bereit, schnell und einfach zu einem alten Entwicklungsstand zurückzukehren. Dies kann aus den verschiedensten Gründen nötig sein.
Subversion arbeitet netzwerkübergreifend wodurch ein verteiltes Arbeiten durch viele Benutzer ermöglicht wird. SVN ist kein Tool welches speziell für die Softwareentwicklung konzipiert wurde, jedoch wird es häufig zu diesem Zweck eingesetzt.
Prinzipiell ist das Vorgehen wie folgt: Es wird ein leeres *repository* auf einem Server angelegt. Dieses kann von beliebig vielen Clients ausgecheckt werden und hat am Anfang die Revisionsnummer 0. Durch das Auschecken holen sich die Clients jeweils eine Kopie des repository und speichern sie lokal auf ihrem Rechner. Auf dieser lokalen Kopie kann anschließend ganz normal mit Dateien und Verzeichnissen gearbeitet werden. Erst am Ende der Arbeit erfolgt clientseitig ein *commit*. Hierbei werden die Dateien und Verzeichnisse die geändert wurden ins Repository hinzugefügt und die Revisionsnummer um eins erhöht. Falls eine Datei durch mehrere Personen gleichzeitig editiert wurde und jeder seine Änderungen per commit einfügen möchte, gibt SVN eine Fehlermeldung aus. Der bzw. die Benutzer müssen in diesem Fall manuell entscheiden, welche der Änderungen als die Neuste behandelt werden soll. Durch die fortlaufende Revisionsnummer ist es ohne Probleme möglich, zu jedem Zeitpunkt zu einer früheren Revision zurückzukehren oder sich alle Änderungen zwischen verschiedenen Revisionen anzeigen zu lassen.
SVN ist prinzipiell ein reines Konsolenprogramm. Jedoch existieren inzwischen zahlreiche graphische Benutzeroberflächen für viele Systeme. Auch für Eclipse gibt es mehrere unterschiedliche Subversion-Plugins, von denen wohl das bekannteste *Subclipse* heißt. Damit ist es möglich das Projekt bzw. das repository in welchem das Projekt gespeichert ist, direkt aus Eclipse zu verwalten.

2.6.4 Trac

Trac ist ein freies webbasiertes Projektmanagementwerkzeug zur Softwareentwicklung.
Zu den Funktionen von Trac zählen eine webbasierte Oberfläche zum Betrachten von Subversion repositories, ein Wiki zum kollaborativen Erstellen und Pflegen von (z. B.) Dokumentation sowie einen *bug tracking system* zum Erfassen und Verwalten von Programmfehlern und Erweiterungswünschen.
Trac ist in Python implementiert und kann, ähnlich wie Eclipse, durch plugins erweitert werden. Primär, jedoch nicht ausschließlich, ist es für den Einsatz in Software-Projekten gedacht. Andere Anwendungsmöglichkeiten wären z.B. die Nutzung als reines Wiki oder *trouble ticket system*.

In unserer Projektgruppe nutzen wir das Trac-System zum einen als Wiki, in dem die Sitzungsprotokolle, wichtige Veröffentlichungen, Links, Anleitungen, Richtlinien etc. gesammelt werden, zum anderen aber auch als Ticket-System, in dem die noch zu erledigenden Arbeiten festgehalten sowie bereits bearbeitete Aufgaben als abgeschlossen abgehackt werden können.

2.6.5 Maven

Maven ist ein sog. build management tool der Apache Software Foundation. Es wurde in Java geschrieben und kann als plugin in Eclipse eingefügt werden. Ziel eines build management tool ist es Programme standardisiert zu erstellen und zu verwalten. Aktuell befindet sich das Maven Projekt in der Version 2.2.x.
Die Grundidee besteht darin, den Programmierer in allen Phasen der Softwareentwicklung soweit wie möglich zu unterstützen und zu entlasten, damit dieser sich auf das Wesentliche konzentrieren kann. Kompilieren, Testen, Verteilen etc. sollen dabei weitgehend automatisiert ablaufen. Auf der anderen Seite soll für diese Tätigkeiten so wenig Konfigurationsaufwand wie möglich anfallen.
Maven speichert alle Informationen über das Projekt in einer XML-Datei (pom.xml) (***p****roject* ***o****bject* ***m****odel*) ab. Diese Datei ist standardisiert und wird beim Ausführen von Maven zuerst syntaktisch überprüft. Genauso standardisiert ist die Verzeichnisstruktur. Hält sich der Entwickler an diese Standardvorgaben, passt Maven selbstständig einen Großteil der Konfigurationsdatei an.
Weiterhin kann man in dieser Datei Softwareabhängigkeiten angeben, welche Maven daraufhin versucht aufzulösen. So lässt sich z.B. JUnit leicht ins Projekt einfügen, wobei die Konfigurationsarbeit zum großen Teil entfällt. Um die Abhängigkeiten auflösen zu können, schaut Maven als erstes in einem lokalen repository nach. Falls dies misslingt, so wird daraufhin versucht die Abhängigkeit auf einem Maven repository im Intra- oder Internet aufzulösen.

2.7 Lizensierung der Ergebnisse

Für die Wahl einer geeigneten Lizenz für die Veröffentlichung der Ergebnisse dieser Projektgruppe („*die Software*“) wurde einstimmig beschlossen, dass die Art der Nutzung, Erweiterung und Verbreitung in jeder Hinsicht uneingeschränkter Natur sein soll. Unter dieser Prämisse wurden einige existierende *Open Source* Lizenzen genauer betrachtet.

Da durch die Bestimmungen des *Copyleft* eine Limitierung oder gar - je nach Ausprägung - Aufhebung der Vereinbarkeit der Software mit anders lizensierter Software einhergehen würde, wurden bereits bei der Kandidatenfindung einige Lizenzen ausgeschlossen. Als wohl prominenteste Beispiele für Lizenzen mit Copyleft seien an

dieser Stelle die *GNU General Public License (GPL)* mit starkem und die *GNU Lesser General Public License (LGPL)* mit schwachem Copyleft genannt.

Die Liste der sinnvoll erscheinenden Kandidaten wurde schließlich gebildet durch die *Apache License*, die *MIT License* und die *Simplified BSD License*.

Sowohl MIT License als auch Simplified BSD License, die sich insgesamt sehr ähnlich sind, bestehen aus einem Copyright Hinweis, einem Haftungs- und Garantieausschluss und der Erteilung der Berechtigung, die lizensierte Software frei nutzen, verändern, anders lizensieren und verbreiten zu dürfen, solange die Verbreitung den Lizenztext und sämtliche Hinweise darauf mit einschließt.

Die Apache License handhabt diese Bestimmungen in vergleichbarer Form, fügt jedoch hinzu, dass mit der Lizenz nicht automatisch die Nutzung von Markenzeichen erlaubt wird. Zudem ist es jeder Person freigestellt, bei der Verbreitung Gewährleistungen (z.B. Support) oder Haftungen einzugehen. Außerdem werden in der Apache License die Weiterverbreitungsbestimmungen präziser ausgeführt und es gibt eine Klausel, die zur Folge hat, dass ohne explizite anderweitige Aussage eingereichte Veränderungen der Software implizit unter derselben Lizenz veröffentlicht werden.

Eine Abstimmung der Teilnehmer dieser Projektgruppe hat zu einer einstimmigen Entscheidung für die Apache License in der aktuellen Version 2.0 geführt. Diese kann unter der URL http://www.apache.org/licenses/LICENSE-2.0.txt eingesehen werden.

3 Seminarphase

Um sich mit Stream-Mining Algorithmen zu befassen und vertraut zu machen, wurden zwei Seminarphasen, jeweils zu Beginn der beiden Semester, veranstaltet. In diesen wurden Vorträge über verschiedene Bereiche gehalten. Die Vorträge der zweiten Seminarphase in Bommerholz werden hier zusammengefasst. Algorithmen, die schließlich auch implementiert wurden, werden in Kapitel 6 näher vorgestellt. Die Ergebnisse der ersten Seminarphase wurden bereits im Zwischenbericht [3] erläutert. Es wird daher an dieser Stelle auf die redundante Darstellung verzichtet.

3.1 Generelles

Das folgende Unterkapitel gibt einen Einblick in die Themen „Zählalgorithmen“ und „Quantilberechnung“ auf Streams. Nähere Informationen zu den implementierten Verfahren finden sich in Kapitel 6.

3.1.1 Zählalgorithmen

Beim maschinellen Lernen fällt schnell auf, dass selbst einfache und triviale Aufgaben, wie z.B. das Zählen, nicht mehr ganz so einfach und trivial sind, wenn sie auf Datenströmen angewendet werden sollen. Natürlich wäre es einfach, wenn wir beliebig viel Speicher zur Verfügung hätten. Dies kommt aber eher selten vor.

Was bringt uns das Abzählen von Elementen? Zum einen bildet dies die Ausgangsbasis von einigen anderen Algorithmen. Das Erstellen von Assoziationsregeln ist so ein Fall. Es gibt eine Reihe von Algorithmen, die auf einer festen Datenbank häufige Elemente bzw. Mengen berechnen, um dann die gewünschten Regeln zur Warenkorbanalyse zu erstellen. Online-Warenhäuser jedoch erstellen sekündlich hunderte von neuen, zu zählenden Transaktionen. Dies macht es schwierig “up-to-date” zu sein. Wer will denn seinen Kunden nur veraltete Bücher empfehlen? Aufgrund der Datenmenge, die ja auch ständig wächst, wird die Schwachstelle der klassischen Zählalgorithmen klar. Es werden Algorithmen benötigt, die live auf dem Strom der Transaktionen zählen.

Aber auch für die uns angestrebte Aufgabe des “Intrusion Detection” ist Zählen hilfreich. Bei der Analyse von IP-Paketen kann durch Zählen bestimmter Pakete und dessen Eigenschaften ein möglicher Angriff erkannt werden.
Ein beliebter Angriff auf ein Netz ist die sogenannte “Denial of Service” Attacke. Diese Attacke sieht vor Dienste, Server und ganze Netzwerke unerreichbar zu machen. Verwendet wird hierbei das TCP Transportprotokoll und der dabei verwendete Drei-Wege-Handshake. Das Protokoll sieht vor, dass der Client an den Server zunächst eine Anfrage zur Sitzungseröffnung schickt - die sogenannten SYN-Pakete. Der Server

antwortet nun mit einem SYN-ACK Paket womit er die Anfrage bestätigt und wartet nun wiederum auf die Bestätigung des Clients mit einem ACK-Paket. Ein Angreifer verschickt nun massenhaft SYN-Paket an den anzugreifenden Server, unterschlägt jedoch das ACK-Paket. Somit ist der Server mit offenen, nicht bestätigten Verbindungen beschäftigt, die er erst nach einem gewissen Timeout aufgibt. Da die Ressourcen eines Servers damit komplett belegt werden können, verweigert er normalen Anfragen den Dienst ("Denial of Service").
Angreifer dieser Art kann man schnell erkennen, indem man für jedes SYN-Paket einer IP bzw. eines Teilnetzes eine eins addiert oder subtrahiert, wenn das entsprechende ACK-Paket erkannt wurde. Wird ein bestimmter Schwellwert überschritten, kann von einem Angreifer ausgegangen werden und die IP-Adresse könnte z.B. von der Firewall geblockt werden.

Wie schon erwähnt kann man nicht nur für bestimmte IP-Adressen zählen, sondern auch für ganze Teilnetze. Hier spricht man von einer hierarchischen Gliederung. Diese kann auch noch weiter verfeinert werden indem man Wissen zu Ports und/oder Empfängeradresse hinzunimmt. Für diese hierarchische Gliederung benötigen wir leicht angepasste Algorithmen.

Die in der Seminarphase vorgestellten Zählalgorithmen waren:

- Lossy Counting
- Sticky Sampling
- CountSketch
- CountMinSketch
- Hierarchical Heavy Hitters

Da Zählalgorithmen recht wichtig sind und die Implementierung nicht sehr komplex ist, wurden alle Algorithmen auch implementiert. In Kapitel 6.1 wird auf die Algorithmen und deren Implementierung weiter eingegangen.

3.1.2 Random Subset Sums - Quantilberechnung

Ein p-Quantil ist genau das Element einer aufsteigend sortierten Liste von N Elementen mit dem Rang $p \times N$. Wenn alle Elemente gespeichert werden ist die Bestimmung des Quantil trivial. In vielen Fällen ist es aufgrund der anfallenden Datenmengen nicht praktikabel.

Bei *Random Subset Sum* (RSS) handelt es sich um ein Verfahren für die Bestimmung von Quantilen. Der Algorithmus arbeitet probabilistisch und garantiert eine Fehlerwahrscheinlichkeit von ϵ und eine Versagenswahrscheinlichkeit von δ.

Eine genaue Beschreibung findet sich im Kapitel 6.2.5

3.2 Häufige Mengen

Beim Finden häufiger Mengen interessiert man sich vordergründig für das gemeinsame Auftreten von *Items* in *Transaktionen*. Die *Häufigkeit* (engl. *frequency*) einer Menge von Items ist definiert, als die Anzahl aller Transaktionen in einer gegebenen Menge von Transaktionen, in denen diese *Items* gemeinsam vorkommen. In der Literatur findet sich unter dem Begriff der Häufigkeit sowohl die absolute, als auch die relative Anzahl dieser Transaktionen. *Häufig* ist eine Menge von Items genau dann, wenn ihre Häufigkeit einen benutzerdefinierten Schwellwert *min support* überschreitet. Die algorithmische Aufgabe besteht nun im Finden sämtlicher *häufiger Mengen* aus einer gegebenen Menge von Transaktionen.
Ein typisches Beispiel hierfür ist die Warenkorbanalyse. Jeder getätigte Einkauf bildet eine Transaktion, jeder im Sortiment des Marktes vorhandene Artikel ein Item. Das finden der häufigen Mengen (gegeben ein Schwellwert min support) in der Menge aller Einkäufe beantwortet hier die Frage, welche Artikel insgesamt häufig alleine (häufige 1-Mengen) und häufig gemeinsam (allgemein häufige n-Mengen) gekauft wurden.

3.2.1 Frequent Pattern Mining

Der Vortrag „Frequent Pattern Mining“gab einen Überblick über den aktuellen Stand der Forschung im Finden häufiger Mengen und stellte zwei neuere Vertreter des Frequent Pattern Minings auf Datenströmen in ihren Ansätzen vor. Als Basis für den Überblicks-Vortrag lag [23] zur Grunde. Als, für die Arbeit auf Datenströmen optimierte, Algorithmen wurden SS-BE/SS-MB: Stream Sequential Pattern Mining with Precise Error Bounds [33] und TSP: Mining Top-K Closed Sequential Patterns [38] präsentiert.

Überblick Zu Beginn wurde der Begriff Frequent Pattern weiter differenziert: Es wird zwischen *frequent itemsets*, *frequent sequentielle pattern* und *frequent structural pattern* unterschieden. Frequent itemsets sind normale häufige Mengen, wie sie zum Beispiel für das Einkaufs-Beispiel wichtig sind. Die häufigen Mengen in frequent sequentielle pattern dagegen besitzen eine Ordnung und sind beispielsweise Artikel, die in einer bestimmten Reihenfolge gekauft wurden. Ganz anders sehen frequent structural pattern aus, bei denen hinter den häufigen Mengen eine gewisse Struktur steckt, die Beziehungen zwischen diesen widergibt.
Wenn in der Anwendung nur frequent itemsets gefunden werden sollen, gibt es zwei Algorithmen, die sich seit Jahren bewährt haben: Apriori und FPGrowth (FPGrowth wird in 3.2.2 vorgestellt). Im naiven Ansatz würde ein Geschäftsführer zum Beispiel alle möglichen Kombinationen an gekauften Artikeln zählen lassen und für diese berechnen, ob sie häufig genug gekauft worden sind. Dies bedeutet allerdings bei einer potentiell großen Menge an häufigen Items eine exponentielle große Potenzmenge

für die Häufigkeiten berechnet werden müssten. Hier setzt nun Apriori an und nutzt die Anti-Monotonie des Mengenverbands aus: Eine Menge kann nur häufig sein, falls alle ihre Teilmengen häufig sind. Durch diese Eigenschaft werden nur noch Mengen gebildet, deren Teilmengen bereits häufig sind und auf Häufigkeit getestet, und nicht die ganze Potenzmenge.
Der Algorithmus ECLAT nutzt auch die Apriori-Eigenschaft aus, verwendet im Gegensatz allerdings ein vertikales Datenformat und speichert pro Item, die Transaktionen in denen es vorkam. Dies erspart Datenbank-Scans und beschleunigt den Algorithmus.
Falls diese Algorithmen allerdings mit sehr großen Datenmengen arbeiten müssen, kommen sie auch an ihre Grenzen. Auswege bieten kondensierte Repräsentationen von frequent itemsets, wie zum Beispiel *closed frequent itemsets*. Ein frequent itemset ist ein closed frequent itemset, falls kein weiteres Item in die Menge aufgenommen werden kann, ohne dass die Häufigkeit der neuen Menge geringer ist, als die der Ausgangsmenge. Algorithmen für closed frequent itemsets auf großen Datenmengen sind CHARM, CLOSET+, AFOPT und FPClose.
Falls die Ordnung der Items wichtig ist, es sich also um *frequent sequentielle pattern* handelt, kann das Problem mittels den Apriori-basierten Algorithmen GSP oder SPADE (benutzt vertikales Datenformat, ähnlich wie ECLAT) gelöst werden, die allerdings, wie Apriori auch, große Kandidatenmengen erzeugen. Um dies zu verhindern, nutzt PrefixSpan die Ordnung der Elemente und teilt sie mittels *projected databases* in Suchräume auf.
Auch bei *frequent structural pattern* gibt es Apriori-basierte Ansätze: AGM, FSG, und der disjoint path-joint algorithm generieren jeweils aus häufige, aber leicht verschiedenen Strukturen, neue Kandidaten, die dann auf ihre Häufigkeit getestet werden. Wie und welche Strukturen zu einem neuen Kandidaten verschmolzen werden, hängt vom jeweiligen Algorithmus ab. Einfachere Wege gehen *pattern-growth*-Algorithmen, die einen Ausgangsgraph einfach zufällig wachsen lassen und dann seine Häufigkeit testen. Dies hat den Nachteil, dass Graphen mehrmals entstehen können, was zum Beispiel der Algorithmus gSpan versucht zu vermeiden, in dem hier neue Strukturen nur an die „rechteste Stelle"des Ausgangsgraphen angehängt werden.

SS-BE/SS-MB: Stream Sequential Pattern Mining with Precise Error Bounds
Wie sich aus dem Namen des Verfahrens lesen lässt, handelt es sich bei SS-BE/SS-MB um Algorithmen, die für *frequent sequentielle pattern* in Streams approximative Schranken für Häufigkeiten ausgeben. SS-BE steht dabei für „Stream sequence miner using bounded error“und SS-MB für „Stream sequence miner using memory bounds“, wobei sich beide Algorithmen in erster Linie im Schneiden der Datenstruktur unterscheiden.
Beide Verfahren teilen als Erstes den Stream in Batches gleicher Größe auf, suchen frequent sequentielle pattern in den Batches und speichern diese in einer lexikographischen Baumstruktur. Die SS-BE Methode garantiert dabei keine false negatives auszugeben. Bei SS-MB ist der Fokus ein Anderer: Hier wird garantiert, dass eine be-

stimmte Speichergrenze nicht überschritten wird. Im Gegenzug kann erst nach der Betrachtung der Daten eine Aussage über false negatives getroffen werden.

TSP: Mining Top-K Closed Sequential Patterns Der TSP-Ansatz arbeitet mit *closed sequential pattern* und fordert vom Nutzer als Parameter nicht eine Mindesthäufigkeit der Items, sondern eine Mindestlänge für häufige Sequenzen. Die Hoffnung besteht da drin, dass es einem Nutzer leichter fällt eine minimale Länge für die gewünschten Sequenzen anzugeben, als den prozentualen Anteil der Häufigkeit eines Items an dem Auftreten aller Items. Intern wird die Mindesthäufigkeit allerdings benutzt und immer auf die Häufigkeit des am wenigsten häufigen closed sequential pattern gesetzt, an welcher Stelle auch die Top-K Idee deutlich wird.
Um möglichst effektiv in der Berechnung zu sein, muss TSP zwei Hauptprobleme lösen: Das erste ist die Frage, wie die (implizite) Mindesthäufigkeit am Schnellsten wächst. Schnelles Wachstum des Parameters verkleinert den Suchraum und beschleunigt das Verfahren. Das zweite Problem ist, wie verifiziert werden kann, dass gefundene *frequent pattern* wirklich *closed* sind. Die Lösung des ersten Problems nennt sich TopSequencesTraversal-Algorithmus, der die Anzahl der aufzubauenden projected databases zu Beginn limitiert, und diese Limitierung mit Steigung der Mindesthäufigkeit immer weiter vernachlässigt. Zur Berechnung der projected databases wird PrefixSpan benutzt. Das zweite Problem löst der ClosedPatternVerification-Algorithmus, der für eine gegebene Sequenz entscheidet, ob diese mit unter die Top-K closed sequential pattern aufgenommen und die Mindesthäufigkeit erhöht werden soll. Dabei muss sinnvoll und effizient entschieden werden, ob die Sequenz aufgenommen oder nicht aufgenommen werden soll und, falls ja, die Mindesthäufigkeit erhöht oder nicht erhöht werden soll.

3.2.2 Finden häufiger Mengen ohne Kandidatengenerierung

Das von Jiawei Han et al. 2000 vorgestellte *FP-growth* [24] gilt als *State of the Art* Batch-Verfahren zum finden häufiger Mengen. Es vermeidet die Kandidatengenerierung und den damit verbundenen kostenintensiven Scan über sämtliche Transaktionen.

Aufbau der Datenstruktur *FP-Tree* In einem Initialisierungsschritt werden sämtliche Transaktionen einmal gescannt und dabei die häufigen 1-Mengen berechnet. Für diese häufigen Items wird eine feste Reihenfolge definiert, indem die Items nach Häufigkeit abfallend sortiert werden. Die Baumdatenstruktur FP-Tree wird mit einer leeren Wurzel initialisiert.
In einem zweiten Scan wird nun für jede Transaktion ein Pfad beginnend direkt unterhalb der Baumwurzel eingefügt, der pro in der Transaktion vorkommendem häufigem Item einen dieses Item repräsentierenden Knoten (gemäß der festgelegten Item-Reihenfolge) beinhaltet und pro Knoten einen Zähler mit eins initialisiert. Sollte der

Anfang des Pfades von der Wurzel aus übereinstimmen mit einem bereits vorhandenen Pfad, wird, solange beide Pfade übereinstimmen, der Zähler jedes gemeinsamen Knotens um eins inkrementiert. Für den ersten Knoten in dem sich beide Pfade unterscheiden wird der verbleibende Pfad als Kind des letzten gemeinsamen Knotens eingefügt. Der Zähler jedes neuen Knotens wird mit eins initialisiert. Der so entstehende FP-Tree extrahiert sämtliche zum finden häufiger Mengen relevanten Informationen aus allen Transaktionen und repräsentiert diese in komprimierter Form.

Extrahieren der häufigen Mengen aus der Datenstruktur *FP-growth* Um aus der so definierten Struktur häufige Mengen zu extrahieren, iteriert FP-growth invers zur festgelegten Reihenfolge über die häufigen Items, vom Seltensten zum Häufigsten. Für jedes häufige Item x wird ein bedingter FP-Tree (*conditional FP-Tree*) zu x konstruiert, nach selbigem Verfahren wie der oben beschriebene FP-Tree, jedoch auf einer eingeschränkten Datengrundlage.
Als Datengrundlage fungieren sämtliche im übergeordneten FP-Tree vorhandenen Präfix-Pfade zu jedem x repräsentierenden Knoten gemeinsam mit dem Zählerstand in jenem Knoten. Diese Datengrundlage wird als *conditional pattern base zu* x bezeichnet. Der Zählerstand repräsentiert die Anzahl der Pfade, die zum x repräsentierenden Knoten führten. Die so konstruierte *conditional pattern base zu* x kann als Menge von x beinhaltenden Transaktionen in der übergeordneten Transaktionsmenge interpretiert werden.
Jedes im conditional FP-Tree zu x vorhandene Item y kam in der ursprünglichen Transaktionsmenge häufig gemeinsam mit x vor, $\{x\} \cup \{y\}$ kann also als häufige Menge gespeichert werden.
FP-growth wird nun rekursiv für sämtliche im conditional FP-Tree zu x häufigen Items y aufgerufen um alle Items zu finden, die gemeinsam mit x und y in der ursprünglichen Transaktionsmenge auftraten. Das Verfahren findet somit rekursiv sämtliche häufigen Mengen, in denen x vorkommt und die noch nicht als häufige Menge identifiziert wurden.
Sind alle Rekursionen für sämtliche häufigen Items x des obersten FP-Trees verarbeitet, wurden alle häufigen Mengen gefunden und das Verfahren ist beendet.

3.3 Klassifikation

Ziel der Klassifikation ist es, Daten in inhaltlich zusammenhängende Gruppen zu gliedern, um so besser mit den jeweiligen Daten arbeiten zu können. Des Weiteren können Daten so besser Eigenschaften zugeordnet werden. Der folgende Abschnitt beschäftigt sich mit Verfahren zur Klassifikation von Daten. Hierbei wird näher auf Entscheidungsbäume, die dazugehörigen Verfahren und auf semi-supervised Lernverfahren eingegangen.

3.3.1 Klassifikation mit Hilfe von Entscheidungsbäumen

Dieser Vortrag beschäftigte sich mit den verschiedenen Ansätzen zur Erstellung eines Entscheidungsbaumes mit dessen Hilfe eingehende Daten klassifiziert werden können. Dabei muss beachtet werden, dass Daten im Nachhinein nicht verändert werden können und dass die Datenreihenfolge beachtet werden muss, da es sich hierbei um High-Speed Data Streams handelt. Diese Art von Daten entsteht beispielsweise bei der Auswertung von Anrufen oder beim Anmelden auf Webseiten. Um diese Menge an Daten schnellstmöglich und effizient auswerten zu können muss auf Entscheidungsbäume zurückgegriffen werden. Das Ziel der Auswertung ist es, anhand einer Funktion direkt neue Daten mit hoher Genauigkeit klassifizieren zu können.
Liegt zu Beginn der Erstellung des Entscheidungsbaums ein klassifizierter, aussagekräftiger Datensatz vor, so ist die Erstellung des Baumes und der dazugehörigen Klassifikationsfunktion mit Hilfe des Kardinalitätskriteriums oder des Informationsgewinns möglich. Bei dem jeweiligen Entscheidungsbaum repräsentiert jeder Knoten ein Attribut. Jede mögliche Belegung eines Attributs ist durch eine Kante dargestellt. Die Blätter des Baumes stellen die Klassifikationswerte da, beispielsweise true oder false. Je höher sich ein Attribut im Baum befindet, umso ausschlaggebender ist dies für das Endergebnis. Wird nun der Baum bei einer Klassifikation eines neuen Datensatzes durchlaufen, so kommt es im jedem Knoten zu einer Abfrage. Vorteil eines Entscheidungsbaumes ist, dass ein großes Problem auf mehrere kleinere Probleme aufgeteilt wird und so nicht die Gesamtkomplexität zu jedem Zeitpunkt betrachtet werden muss. Ein idealer Baum ist möglichst klein und die einzelnen Knoten sind gleichmäßig verteilt.
Während des Vortrags wurden mehrere Algorithmen zur Erstellung von Entscheidungsbäumen betrachtet und erläutert. Auf die folgenden wurde genauer eingegangen:

1. ID-3
2. C4.5
3. CART
4. Hoeffding Bäume

Dabei handelt es sich um klassische Entscheidungsbäume. Hierbei ist die maximale Größe des Baumes durch den Arbeitsspeicher beschränkt. Des Weiteren müssen sich alle Daten von Beginn an im Hauptspeicher befinden.
Der ID-3-Algorithmus ist ein top-down Vorgehen. Dabei werden zunächst die Eigenschaften der Wurzel festgelegt. Dies ist das Attribut mit dem höchsten Information Gain. Die einzelnen Eigenschaften für die Knoten werden mit Hilfe von Greedy Search festgelegt. Leider hat das ID-3-Verfahren noch einige Schwachstellen. Es treten Probleme im Bereich Overfitting/ Underfitting, bei der Gruppierung von Attributen und bei der Bearbeitung von Datensätzen in denen einzelne Attribute fehlen, auf. Bei

C4.5 handelt es sich um eine Weiterentwicklung des ID-3 Algorithmus. Der Unterschied bei diesem Verfahren ist, dass bei der Erstellung eines Entscheidungsbaumes zunächst nur die Hälfte der Daten betrachtet wird. Mit der anderen Hälfte der Daten wird anschließend der Baum durchlaufen. Hierbei wird geprüft, ob mit Hilfe des Baumes dieselben Klassifikationen erreicht werden können. Ist dies nicht der Fall so wird nur der ausschlaggebende Teil des Baumes angepasst. Um den Problemen die ID-3 besitzt aus dem Weg zu gehen, wurde festgelegt, dass Trainingsdaten in denen Attributbelegungen fehlen, nicht betrachtet werden bei der Erstellung eines Baumes. Bei der Klassifikation von Datensätzen mit fehlenden Attributbelegungen besteht das Ergebnis aus Wahrscheinlichkeiten, die die einzelnen Klassifikationen darstellen. Im Gegensatz zu ID-3 kann nun auch eine Gruppierung von nummerischen Werten vorgenommen werden. CART steht für Classification and Regression Tree. Bei dem Verfahren können lediglich Binärbäume erzeugt werden. Um dies zu realisieren werden die Attributbelegungen des jeweiligen Attributs in Kandidatenmengen geteilt. Mit Hilfe von Hoeffding Bäumen können alle Beispiele in konstanter Zeit verarbeitet werden. Des Weiteren wird jedes Beispiel nur einmal betrachtet und auch nur so lange bis ein neues Beispiel ankommt. Bei Hoeffding Bäumen werden die einzelnen Attribute über die Hoeffding-Schranke errechnet.
Um den Eigenschaften eines Daten Streams besser gerecht zu werden, verwendet man Very Fast Decision Trees (VFDT). Diese bauen auf den Hoeffding Bäumen auf, ermöglichen es jedoch mehrere zehntausend Beispiele pro Sekunde zu bearbeiten. Vorrausetzung ist jedoch, dass die einzelnen Beispiele gleichmäßig verteilt sind. Ein wichtiger Vorteil im Vergleich zu ID-3 und C4.5 ist, dass die Datenmenge nicht mehr nur auf den Hauptspeicherplatz beschränkt ist. Eine Erweiterung der VFDT bieten die Concept-adapting Very Fast Decision Trees (CVFDT) . Die Performance und die Geschwindigkeit ist dieselbe, sie sind jedoch auf große Datenmengen mit hoher Datenfrequenz spezialisiert. Ein weiterer Vorteil ist, dass zu jedem Split in einem Knoten die Wahrscheinlichkeiten zu anderen Splits mit gespeichert werden. Dies hat den Vorteil, dass bei Änderungen nicht erst der jeweilige beste Split errechnet werden muss, sondern das Verfahren kann auf diesen direkt zugreifen ohne zusätzlichen Rechenaufwand.
Das Fazit ist, dass die Verfahren VFDT und CVFDT die idealsten unter den betrachteten Verfahren sind. Jedoch muss hier die Einschränkung beachtet werden, dass die Attributmenge der Daten nur eine geringe Vielfalt aufweisen darf.

3.3.2 Semi-supervised Clustering

Gegenüber unüberwachten Lernverfahren haben *semi-supervised* Lernverfahren den Vorteil, vorhandenes Hintergrundwissen über die vorliegenden Daten zu nutzen. Beim Clustering fließen solche Zusatzinformationen in Form von *constraints* ein. Wenn z.B. bekannt ist, dass zwei Elemente des Streams nicht in eine gemeinsame Klasse gehören, wird als constraint ein *cannot link* gesetzt. Analog werden Elemente die sicher

in einer Klasse auftreten durch einen *must link* verknüpft. Diese Informationen können beim semi-supervised Clustering u.A. verwendet werden um gute initiale Cluster zu finden.
Um ein Beispielverfahren für ein semi-supervised Lernverfahren zum Clustern von Daten im Detail darzustellen wurde im weiteren Verlauf des Vortrags das Verfahren **SmSCluster**[32] thematisiert. Es handelt sich hierbei um ein online Lernverfahren, das den Datenstrom in Blöcke aufteilt. Diese Blöcke wiederum sind in Test- und Trainingsdaten zu unterteilen. Da es sich um ein semi-supervised Lernverfahren handelt, müssen die Trainingsdaten einen bestimmten Anteil an klassifizierten Daten enthalten. Es kann jedoch bereits dann mit guten Ergebnissen gerechnet werden, wenn dieser Anteil klein ist. Die Testdaten des Datenstroms können jederzeit verarbeitet werde (ein bestehendes Modell vorausgesetzt), so dass nur die Trainingsdaten gepuffert werden müssen bis sie einen ausreichenden Anteil klassifizierter Daten beinhalten.

$$O_{k-means} = \sum_{i=0}^{k} \sum_{x_j \in C_i} \|x_j - \mu_i\|^2 \tag{1}$$

Um die Daten eines Blocks zu gruppieren, wird der *k-means* Algorithmus verwendet. Der k-means Algorithmus wählt Punkte so aus, dass der Abstand aller Clustermitglieder zum Clusterzentrum minimal für alle Cluster ist (Formel 1). Der einfache k-means-Algorithmus wird um eine Straffunktion erweitert. Anhand der klassifizierten Daten kann für jedes Cluster eine Hauptklasse definiert werden. Dies ist die Klasse der größten Gruppe von Punkten im Cluster. Durch die Straffunktion werden Punkte eines Clusters „teurer“, die nicht zur Hauptgruppe des Clusters gehören. Die resultierende Zielfunktion gewichtet die Summe der Abstände (Formel 1) mit der Unähnlichkeit und der Entropie des Clusters. Im Allgemeinen resultiert die Minimierung dieser Zielfunktion (Formel 2) zu „reineren“ Clustern.

$$O_{SmS} = \sum_{i=0}^{k} \left(\sum_{x_j \in C_i} \|x_j - \mu_i\|^2 \right) \cdot (1 + Ent_i \cdot Dis_i) \tag{2}$$

Zur optimalen Lösung müsste jede mögliche Kombination berechnet und anschließend das Clustering mit minimalen Kosten ausgewählt werden. Da der Aufwand hierfür im Allgemeinen allerdings zu hoch ist und es sich ein Minimierungsproblem mit unvollständigen Daten (es sind weder die Mittelpunkte noch die Hauptgruppen der Cluster bekannt) bietet es sich an dieses Problem mit dem *expectation-maximization*-Algorithmus zu lösen. Für das Clustering iteriert der EM-Algorithmus über folgenden drei Schritten:

- **Initialisierung:** Es werden k klassifizierte Punkte als Clusterzentren als Ausgangspunkt gewählt.
- **E-Schritt:** Für jeden weiteren Punkt wird die Zielfunktion für jede mögliche Cluster-Zuordnung berechnet. Anschließend wird die Zuordnung gewählt, die

die Zielfunktion minimiert.

- **M-Schritt:** Nachdem alle Punkte einem Cluster zugeordnet wurden müssen die Clusterzentren für die nächste Iteration berechnet werden.

Für die Iterationen wird der *iterative conditional mode*-Algorithmus[5] verwendet. Die Reihenfolge der Punkte wird vor jeder Iteration zufällig angeordnet und sequentiell abgearbeitet. Der EM-Algorithmus terminiert, wenn die gefundene Zuordnung konvergiert, d.h. wenn sich die Zugehörigkeit der Punkte zu ihren Clustern nicht mehr ändert. Der ICM-Algorithmus garantiert ein lokales Minimum zu finden. Die Anzahl der Iterationen ist stark von der Wahl der initialen Clusterzentren abhängig. Als Startwerte wird das *k-dominating set* der Menge bestimmt [27].

Nachdem der Algorithmus die Trainingsdaten wie oben beschrieben gruppiert hat, werden die wesentlichen Informationen aus diesem Clustering extrahiert und in *Micro-Clustern* gespeichert. Dadurch nimmt der Speicherbedarf ab, weil nicht alle Punkte selbst gespeichert müssen. Ein Micro-Cluster enthält folgende Werte:

- Die Anzahl n der Punkte im Cluster.
- Die Anzahl L der klassifizierten Punkte im Cluster.
- Einen C-dimensionalen Vektor. Jede Dimension enthält die Anzahl der Punkte einer Klasse im Cluster.
- Das Zentrum μ des Clusters
- Einen Vektor für die Summe aller Vektoren des Clusters.

Die so entstehende Zusammenfassung des Clusters enthält ausschließlich additive Werte und ermöglicht es, Cluster auf einfachste Art und Weise während des Updates zu verschmelzen.
Der anschließende Teil des Vortrags hat sich der Update-Funktion von SmSCluster gewidmet. Sobald das Ensemble seine maximale Anzahl von L Micro-Clustern erreicht hat, muss für jedes neue Micro-Cluster ein Platz im Ensemble frei gemacht werden. Dies geschieht, indem zwei Micro-Cluster zu einem verschmolzen werden um den Informationsverlust gering zu halten. Hierzu werden zwei Cluster ausgewählt, die dieselbe Hauptklasse (d.h. die Klasse der größten Gruppe von Punkten in einem Cluster) und minimalen Abstand besitzen.
Da im Vorfeld nicht bekannt ist, welche Klassen in den Trainingsdaten auftreten können, muss in der Update-Funktion ein weiterer Fall betrachtet werden. Falls ein neuer Micro-Cluster Beobachtungen aus Klassen enthält, die zuvor nicht aufgetreten sind, werden Testfehler wahrscheinlicher. Um dies zu umgehen wird die neue Klasse mit einer im Vorfeld definierten Wahrscheinlichkeit in andere Cluster eingefügt.

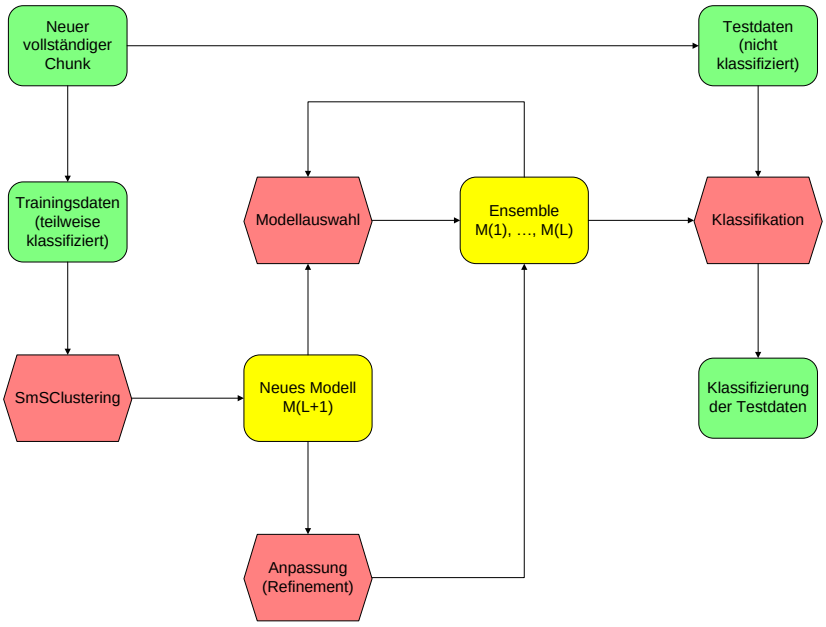


Abbildung 6: Der Ablauf des SmSCluster Verfahrens im Überblick

Nachdem nun die Modellbildung ausführlich erörtert wurde, konnte der vorletzte Teil des Vortrags das Testen von unbekannten Daten behandeln. Das SmSCluster-Verfahren berechnet zu diesem Zweck mit jedem Micro-Cluster des Ensembles eine Klassifizierung. Die Ergebnisse werden geeignet miteinander verrechnet und ergeben so die Ausgabe für ein zu testendes Datum.

Der Vortrag wurde durch die experimentellen Ergebnisse der Autoren von SmSCluster abgerundet. Es wurde aufgezeigt, welche Parameter Einfluss auf die Güte der Lösung sowie die Laufzeit des Algorithmus haben.

3.4 Unüberwachtes Lernen

3.4.1 Clustering auf Streams

Unter *Clustering* versteht man das unüberwachte Auffinden von Gruppierungen ähnlicher Elemente in einer Menge von Datenpunkten. Dies gestaltet sich auf Streams als schwierig, da eine Speicherung aller Datenpunkte aufgrund der Unendlichkeit des Streams nicht realisierbar ist [18]. Daher ist keine ganzheitliche Betrachtung aller Datenpunkte möglich. Stattdessen wird auf statistischen Daten, kumulierten Werten oder ausgewählten Datenpunkten gearbeitet. Trotzdem existieren für Streams eine

Reihe an Verfahren, die zumindest approximativ gute Clusterings auf Datenströmen durchführen. Eine Auswahl dieser Verfahren soll im Folgenden kurz erläutert werden.

STREAM sammelt Datenpunkte bis eine definierte Menge von Datenpunkten erreicht ist und führt auf dieser *chunk* genannten statischen Menge ein herkömmliches K-Median Verfahren aus. Die errechneten Clusterzentren werden gespeichert, alle übrigen Datenpunkte hingegen werden verworfen. Mit der Entwicklung des Datenstroms existieren so $\#Clusterzentren\ pro\ chunk \cdot \#chunks$ viele Clusterzentren, auf denen ein *high level clustering* durchgeführt werden kann. [35]

CluStream ist ein dichtebasiertes Clustering-Verfahren. Es besteht aus zwei Phasen: online werden permanent statistische Daten über die eingehenden Datenpunkte erfasst und in ein spezielles Format, den sogenannten *MicroCluster*, überführt. In regelmäßigen Abständen werden *snapshots* dieser MicroCluster erstellt und nach dem Muster *pyramidal time frame* gespeichert, welches für erst kurz zurückliegende Daten eine hoch aufgelöste und für ältere Daten eine geringer aufgelöste Verteilung der MicroCluster bietet. In einer unabhängigen Offlinephase kann über die snapshots der MicroCluster für einen vom aktuellen Zeitpunkt aus beliebig weit zurückreichenden Zeitraum ein high level clustering durchgeführt werden. [1]

HPStream ist eine Modifikation von *CluStream*. Qualifiziert sich durch Techniken zur Reduzierung der Dimensionalität von Daten jedoch besonders für hochdimensionale Daten.

D-Stream ist eine Verbesserung von CluStream. Im Gegensatz zu CluStream kann es

- Cluster beliebiger Form finden. Die Verwendung des k-Means Ansatzes bei CluStream hat zur Folge hat, dass die dort gefundenen Cluster immer Kugeln sind.
- Outlier (Noise) behandeln.
- eine beliebige Anzahl von Clustern finden. Bei *CluStream* muss die Anzahl der Cluster im Vorfeld eingestellt werden.
- *concept drift* besser erkennen, da die Dichte alte Datenpunkte durch eine Verfallsfunktion stetig reduziert wird.

Wie CluStream arbeitet auch *D-Stream* mit einer Online- und einer Offline- Komponente. Die Online-Komponente mappt die ankommenden Datenpunkt auf sogenannte *grids* in einem Raster. Die Offline-Komponente fasst dann die belegten grids unter Berücksichtigung ihrer Dichte zu Clustern zusammen. Das Verfahren wurde im Rahmen der Projektgruppe implementiert. Eine ausführliche Dokumentation findet sich im Kapitel 6.5.2. [12]

DenStream ist eine Modifikation von D-Stream. Das Verfahren weißt im wesentlichen die selben Vorteile gegenüber CluStream auf. Der Unterschied zu D-Stream liegt in der Verwendung von *c-micro-clustern* anstelle von grids.[8]

IncrementalDBSCAN bildet Cluster aufgrund der Dichte der Datenpunkte in der Nachbarschaft eines Datenpunktes.

RepStream arbeitet mit zwei Graphen, in denen erstens Verknüpfungen und die zweitens die Dichte der betrachteten Datenpunkte gespeichert werden. Datenpunkte, die in beiden Graphen besonders ausgeprägt sind, werden als Repräsentanten ausgewählt. Diese Repräsentanten werden nach Nützlichkeit sortiert in einem repository gehalten.

BIRCH ist ein hierarchisches Clusteringverfahren, das versucht so viel Arbeitsspeicher eines Rechners wie möglich auszunutzen, um eine möglichst genaue Hierarchie der Cluster zu erzeugen. Stößt das Verfahren dabei an die Grenzen des Speichers (was auf Streams in der Regel der Fall ist), wird die Hierarchie umgebaut und vergröbert, um Speicher wieder freigeben zu können. Auf BIRCH wird im Abschnitt 6.7.1 genauer eingegangen.

3.4.2 ClusTree

In Kapitel 3.4.1 wurden bereits *K-Median Clusteringverfahren* vorgestellt. Ein Kritikpunkt an diesen Verfahren stellt die Festlegung auf k verschiedene Clusterzentren dar. Diese Festlegung ist unter vielen Fachleuten umstritten und gilt als zu statisch. Daher wurden alternative Techniken entwickelt, die eine flexible Anzahl von Clusterzentren unterstützen. Ein solches Verfahren ist *ClusTree* [29].

Die Idee von ClusTree ist die Darstellung einer Menge von hierarchischen Clustern innerhalb einer Baumstruktur. Jeder Knoten des Baumes ist dabei ein eigenes Cluster-Zentrum und hält statistische Informationen zur Berechnung des Cluster-Mittelpunkts bereit. Je tiefer ein Knoten innerhalb des Baums liegt, umso spezifischer ist der dargestellte Cluster. Während die Wurzel des Baums alle Objekte enthält und somit die gröbste Darstellung eines Clusters liefert, umfasst ein Blatt des Baums nur eine geringe Anzahl an Elementen innerhalb des Baums. Dadurch wird eine hierarchische Sichtweise auf die Menge der Cluster ermöglicht, deren Granularität je nach Anwendungsfall angepasst werden kann.

ClusTree ist ein auf Streams optimiertes Anytimeverfahren. Zu jedem Zeitpunkt innerhalb des Lernintervalls kann also ein valides Modell abgerufen werden. Spezialisiert ist der Algorithmus dabei auf wechselnde Geschwindigkeiten innerhalb eines Streams. Ausgangssituation für diese Spezialisierung ist die Annahme, dass es durch die wechselnden Geschwindigkeiten innerhalb des Streams und eine Menge von schnell aufeinander folgenden Objekten dazu kommt, dass der Baum während

des Lernverfahrens nicht vollständig durchlaufen werden kann. In diesem Fall werden Objekte zunächst in einem Knoten abgelegt und gelangen später als so genannte *Hitchhiker* unter geringem Geschwindigkeits-Verlust in ein Blatt des Baums.

3.5 Verteilte Verfahren

3.5.1 Zusammenführung von verteilten Streams

Wenn man mit verteilten Streams arbeitet, kommt es häufig vor, dass man Daten aus unterschiedlichen Quellen gleichzeitig verarbeiten will, um z.B. Zusammenhänge zwischen den einzelnen Datensätzen herstellen zu können. Hierfür müssen die unterschiedlichen Streams mit einander verschmolzen werden. Häufig beinhalten die Streams gemeinsame Features, wie z.B. einen Zeitstempel von unterschiedlichen Log-Dateien. Dadurch wird es möglich die beiden Streams mit Hilfe eines *sliding window* Verfahrens über den so genannten *common key* zu verschmelzen.

Als Maß für die Vollständigkeit des Merges nutzt man den so genannten gleitenden Mittelwert:

$$\delta_{m,n} = 1 - \frac{\sum_{i \in [(n-m+1),n]} \kappa_i}{m}, m > 0, n \geq m$$

m =Anzahl der Fenster die bearbeitet werden
n = Letzes Fenster welches in den gleitenden Durchschnitt einfließt
κ_i = erfolgreiche *merge*s im Fenster i

Man unterscheidet grundsätzlich zwischen Best Effort δ-Merges und Best Effort ϵ, δ-Merges.

Best Effort δ-Merges bezeichnen die Menge der Merge-Verfahren, für die der gleitender Mittelwert kleiner als ein ein frei gewähltes δ ist, wobei m > 0; n $\geq$ m.

Best Effort ϵ, δ-Merges enthalten alle Merge-Verfahren der δ-Merges erlauben aber zusätzlich einen gewissen fest gesetzten Fehler ϵ.

Im Folgenden werden die Best Effort δ-Merges nicht weiter betrachtet. Als Vertreter für die Klasse der Best Effort ϵ, δ-Merges werden die Grundlagen des **GCM-** und des **RTM-Algorithmus** vorgestellt.

Beiden Algorithmen gemein ist, die Verwendung von sliding windows. Beide Streams befüllen jeweils ein sliding window der Größe N. Dabei unterscheidet sich allerdings die Auswahl der Daten für die *windows*, je nach Algorithmus. Jedes sliding

window verwaltet zusätzlich einen Zeiger auf den aktuellen Datensatz. Für dieses Beispiel verwenden wir zwei Streams A und B für die jeweils der Zeiger a bzw. b verwaltet werden. Zusätzlich müssen noch zwei Werte N_a und N_b verwaltet werden, die die minimale Anzahl neu einzufügender Daten definieren.

GCM Algorithmus

1. Für das Befüllen der sliding windows werden die nächsten N Daten der Streams genommen.
2. Die Datensätze in den *windows* werden nach dem common key sortiert.
3. Der *merge* wird durchgeführt:
   a) Wenn der Wert des common key des Datensatzes auf den b zeigt größer ist als der Wert von $a + \epsilon$, also $[b]_V > [a + \epsilon]_V$
   $\Rightarrow$ setze Zeiger a auf den nächsten Datensatz
   b) Wenn der Wert des common key des Datensatzes auf den b zeigt im Bereich des Wertes von $a \pm \epsilon$ ist, also $[b]_V \in [a \pm \epsilon]_V$
   $\Rightarrow$ verschmelz beide Datensätze und setze a auf den kleinsten Datensatz dessen Wert größer als $\epsilon + lastMergedValue$ ist
   c) Wenn der Wert des common key des Datensatzes auf den a zeigt größer ist als der Wert von $b + \epsilon$, also $[a]_V > [b + \epsilon]_V$
   $\Rightarrow$ setze Zeiger b auf den nächsten Datensatz
4. Neue Daten werden in die sliding windows eingefügt, wenn ein Zeiger den N'ten Datensatz erreicht.
   Nehmen wir an Zeiger a erreiche zuerst den N'ten Datensatz:
   a) wurden M Datensätze verschmolzen setzt man:
   $l_a = max(M, N_a); l_b = max(M, N_b)$
   b) es werden l_a neue Datensätze in *window* A geladen.
   Als erstes werden die Slots überschrieben, die erfolgreich verschmolzen wurden, danach die ersten $N_a - M$ Datenslots

Danach beginnt man wieder bei Schritt 2.

RTM Algorithmus Dieser Algorithmus arbeitet ähnlich wie der GCM-Algorithmus. Es wird allerdings ein R-Tree als Datenstruktur vorausgesetzt. Diese kann effizient angefragt werden um die passenden Daten zum verschmelzen auszuwählen.

1. das sliding window A wird mit N Daten des Streams A gefüllt.

2. bestimme das Maximum und Minimum des common key von den in window A vorliegenden Daten und lade N Daten gemäß der berechneten Grenzen in window B
3. sortiere und verschmelze die Daten wie bei GCM
4. füge neue Daten nach folgendem Schema ein:
   a) wenn Zeiger a zuerst den N'ten Datensatz erreicht:
   $\Rightarrow$leere window A und beginne bei Schritt 1
   b) wenn Zeiger b zuerst den N'ten Datensatz erreicht:
   $\Rightarrow$leere window B und lade die neuen Daten basierend auf der symmetrischen Differenz zwischen den alten Daten aus window B und den aktuellen Daten aus window A
   $\Rightarrow$ wenn die Anfrage nicht leer ist, fülle window B mit N Daten des Ergebnisses
   $\Rightarrow$ wenn die Anfrage leer ist, leere window A und gehe zu Schritt 1

Beide Algorithmen können sehr leicht implementiert werden. Dadurch lässt sich eine effiziente Verschmelzung zweier oder mehrerer Streams realisieren.

3.5.2 Bayes'sche Netze aus verteilten Datenströmen

Das Bayes'sche Netz ist ein Graphenmodell, das der Darstellung von wahrscheinlichkeitstheoretischen Zusammenhängen zwischen Zufallsvariablen dient und Vorhersagen über die Wahrscheinlichkeit des gemeinsamen Auftretens unterschiedlicher Ereignisse ermöglicht. Häufig findet es Anwendung in der Repräsentation von unsicherem Wissen.

Ein Bayes'sches Netz wird beschrieben durch einen gerichteten azyklischen Graphen $G = (V, E)$. Dabei stellen die Knoten $v \in V$ die Zufallsvariablen dar, während die gerichteten Kanten $\langle v_i, v_j \rangle \in E$ bedingte Abhängigkeiten $v_j|v_i$ zwischen diesen Zufallsvariablen repräsentieren. Dabei ist jede Zufallsvariable v_i nur von ihren unmittelbaren Vorgängern abhängig.

Beim Vorgang des Lernens eines Bayes'schen Netzes aus einer gegebenen Datenmenge wird grundsätzlich unterschieden zwischen dem Lernen der Struktur des Netzes - also der Menge der Knoten und deren Verknüpfungen untereinander - und dem Lernen der bedingten Abhängigkeiten.

Für den Fall eines einzelnen Datenstroms, bzw. der Bündelung mehrerer Datenströme zu einem einzigen, betrachtet Friedmann in [16] drei Varianten:

- Beim *naiven* Ansatz wird jeder unterschiedliche Datenpunkt zusammen mit der Anzahl seiner bisherigen Vorkommen im Datenstrom gespeichert. Bei jedem neuen Datenpunkt wird ein *Batch*-Lerner verwendet, um ein neues Modell für die bisherigen Daten zu errechnen. Qualitativ liefert dieser Ansatz das beste Ergebnis, der Speicheraufwand ist jedoch exponentiell in der Anzahl der Variablen und damit nicht praxistauglich.
- Beim MAP (**m**aximum **a**-posteriori **p**robability) Ansatz wird im Gegensatz zum naiven Ansatz nicht jeder Datenpunkt, sondern nur das aktuelle Modell gespeichert. Dieses stellt eine Zusammenfassung der bisherigen Datenpunkte dar und kann damit als a-priori Wissen für den nächsten Lerndurchlauf, zusammen mit den neu beobachteten Datenpunkten verwendet werden. Zwar ist diese Herangehensweise speichereffizient, es besteht jedoch die Gefahr, dass mit der Zeit eine Art fester Zustand erreicht wird, von dem sich neue Modelle nicht mehr entfernen.
- Die *inkrementelle*, auf MAP basierende Variante speichert ebenfalls das aktuelle Modell. Darüber hinaus werden jedoch einige Daten verwaltet, um aus dem gespeicherten Modell ähnliche, bereits verworfene Modelle rekonstruieren zu können. Bei einem neuen Datenpunkt wird dann das aktuelle Modell erweitert, es wird jedoch zusätzlich überprüft, ob eines der ähnlichen Modelle besser wäre.

In allen drei Fällen bietet es sich an, nicht jeden neuen Datenpunkt des Datenstroms einzeln zu bearbeiten, sondern erst eine definierte Anzahl an Datenpunkten zu sammeln und diese dann gemeinsam zu bearbeiten.

Bei verteilt anliegenden Datenströmen scheint es erstrebenswert, diese nicht in einer zentralen Stelle gebündelt als einen einzigen Datenstrom zu behandeln. Vielmehr möchte man die verteilte Rechenleistung nutzen und Kommunikation minimal halten. Darüber hinaus macht eine Bündelung die Wahrung der Vertraulichkeit von Daten schlichtweg unmöglich.
Unter anderem aus diesen Gründen betrachtet Chen in [11] einen anderen Ansatz: Für jeden Datenstrom wird ein eigenes Bayes'sches Netz verwaltet und lediglich aus Stichproben werden diese lokalen Modelle in einer zentralen Stelle miteinander in Verbindung gesetzt.
Das Lernen der Struktur eines Bayes'schen Netzes gelingt bislang jedoch nicht, weshalb eine Vorlaufphase erforderlich ist, in der durch einen Beispieldatensatz oder Expertenwissen die Struktur des gesamten Netzes festgelegt werden muss. Ist diese einmal gegeben, können fortan die bedingten Abhängigkeiten kontinuierlich angepasst werden.

3.5.3 Verteiltes Clustering

Beim Clustering auf verteilten Streams gibt es diverse Probleme die es zu meistern gilt. Zum einen müssen die bereits angesprochenen Speicherplatzprobleme bei Stream Mining Verfahren berücksichtig werden: wir haben keine Möglichkeit den kompletten Stream zu speichern und laufen somit Gefahr wichtige Daten zu verlieren. Zum anderen müssen zeitkritische Entscheidungen getroffen werden: die Daten können nur ein einziges Mal betrachtet werden und es ist bei diesem Schritt entscheidend, dass der Verarbeitungsschritt nicht zu viel Zeit in Anspruch nimmt, da sonst Informationsverluste auftreten können. Beim verteilten Clustering auf Streams kommt zusätzlich hinzu, dass es immer einen zentralen Knoten geben muss, der sich um die Administration des ganzen Netzwerks kümmert. Hierbei gibt es das Ziel, dass eben dieser Knoten nicht die gesamte Netzwerkkommunikation abwickelt, da sonst der Aspekt des verteilten Rechnens zunichte gemacht wird und keine Vorteile gegenüber der lokalen Lösung existieren.
Es muss also möglich sein, die Kommunikation zwischen den einzelnen Knoten auf ein Minimum zu reduzieren. Jedoch sollte man bei diesem Vorhaben unter keinen Umständen die Güte des Clusterings vernachlässigen. Die gegebene Situation ist in Abbildung 7 grafisch dargestellt.
Der erster naive Ansatz wäre der, dass jeder Knoten, zu jedem Zeitpunkt t, den zen-

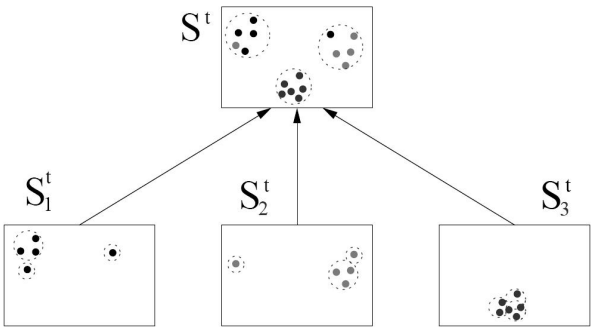
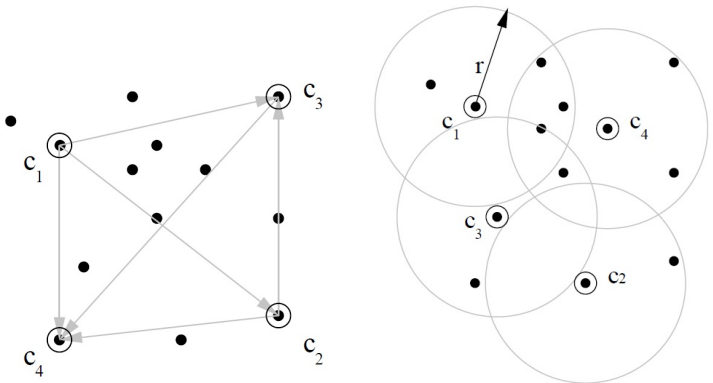


Abbildung 7: Verteiltes Clustering

tralen Knoten mit seinen aktuellen Punkten aktualisiert. Hierbei besteht jedoch das Problem, dass ein *Bottleneck* entsteht und sowohl die Kommunikation wie auch die Verarbeitung ins Stocken geraten.
Der zweite naive Ansatz besteht darin, periodisch alle τ Sekunden eine Aktualisierung seiner Datenpunkte zum zentralen Knoten zu schicken. Auch mit diesem Ansatz wäre das Kommunikationsproblem nicht gelöst. Außerdem kann man nicht vorhersagen wie groß die Änderungen zwischen den einzelnen Intervallen sind.
Im Folgenden werden zwei bessere Ansätze präsentiert, die für diesen Zweck konzipiert wurden. Zum einen handelt es sich dabei um den *Furthest Point*-Algorithmus und zum anderen um den *Parallel Guessing*-Algorithmus . Beide Verfahren werden erläutert sowie gegenüber gestellt. Bei beiden Verfahren handelt es sich um sog. α-

Approximative Algorithmen. Das bedeutet, dass die Kosten für den Algorithmus maximal das α-fache vom optimalen Algorithmus entfernt sind.
Um den Furthest point-Algorithmus am besten zu verstehen schaut man parallel zum Lesen die Abbildung 8a an. Zu Beginn wählt man zufällig einen beliebigen Punkt als Mittelpunkt - in der Abbildung wäre dies C_1. Die weiteren Schritte bestehen darin die i + ersten Center, die eine maximale Distanz zu den anderen Mittelpunkten haben, zu finden und auszuwählen. Da man im Vorfeld die Anzahl der Cluster festlegt, findet dieser Algorithmus nach k Iterationen diese und endet. Als Abstandsmaß kann hierbei z.B. der euklidische Abstand genutzt werden. Wichtig ist lediglich, dass ein eine Distanzfunktion gibt, mit der sich die Abstände messen lassen.
Der Parallel Guessing-Algorithmus hat eine geringfügig schlechtere α-Approximation



(a) Furthest Point (b) Parallel Guessing

Abbildung 8: Furthest Point & Parallel Guessing

bietet jedoch den Vorteil, dass der Speicherverbrauch unabhängig von der Anzahl der Datenpunkte ist. Weiteres wichtiges Kriterium ist, dass nur ein Durchlauf über die Daten durchgeführt werden muss. Auch hier schaut man sich am besten die Abbildung 8b an, um im Folgenden die einzelnen Schritte besser zu verstehen. Man geht von der Annahme aus, dass der Radius bereits bekannt ist. Wie beim vorherigen Algorithmus wählt man hierbei den ersten Mittelpunkt beliebig. Im nächsten Schritt wird für jeden Datenpunkt die Distanz zu diesem Mittelpunkt berechnet und hierbei das Minimum betrachtet. Wenn dieser größer als der festgelegte Radius ist, entsteht somit ein neuer Mittelpunkt. Falls nicht, wird dieser Datenpunkt zum jeweiligen Mittelpunkt hinzugefügt.

4 Design & Architektur

Das folgende Kapitel gibt einen Einblick in das Framework, seine Architektur und das Design. Zunächst gehen wir auf das **Konzept** ein. Es wird geklärt wie wir Stream-Mining Algorithmen auf das Framework abbilden und welche Terminologie wir dabei verwenden. Daraufhin gehen wir auf die **Anwendung** ein. Es wird gezeigt wie ein Benutzer das Framework verwenden kann, d.h. wie man einen Algorithmus zusammenstellen und ablaufen lassen kann. Erst im darauf folgenden Kapitel beschäftigen wir uns damit, wie man Algorithmen und Knoten entwickelt. Zuletzt werden die technischen Details erklärt, die notwendig sind, um den Kern des eigentlichen Frameworks weiterzuentwickeln.

4.1 Konzept

Das Framework ist für die Planung und Durchführung von **Stream-Mining** Algorithmen gedacht. Dabei werden unendliche Datenströme aus Quellen gelesen und kontinuierlich verarbeitet. Beispielsweise bilden die Einträge in Log-Files von mehreren Webservern einen prinzipiell unendlichen Strom an zu verarbeitenden Daten. Jeder Eintrag bildet eine Einheit, die wie in einer Pipeline mehrere Verarbeitungsstationen durchläuft. Sind diese Stationen generisch konzipiert, können sie in einem anderen Kontext wiederverwendet werden. Dies erhöht die Planbarkeit und Wartbarkeit solcher Algorithmen.

Solche Verarbeitungsstationen, die untereinander verbunden sind und einen kontinuierlichen Datenstrom bearbeiten, nennen wir **Knoten**. Ein **Knoten** ist die kleinste funktionelle Einheit, die Daten bearbeitet. Die Daten, die als Einheit zu bearbeiten sind, werden **Event** genannt. Ein Knoten erhält also ein Event, das z.B. eine Webserveranfrage aus dem derzeitigen Datenstrom beinhaltet und arbeitet auf diesem. Das bedeutet beispielsweise, dass er die Daten vorverarbeitet, aggregiert, auf ihnen ein Modell lernt, es auswertet, etc.

Ein Event ist dabei eine Sammlung von verschiedenen Daten. Das bedeutet, dass ein Knoten zu einem Event Daten hinzufügen, Daten modifizieren oder gar Daten entfernen kann. Nach dieser Verarbeitung wird dieses Event nun typischerweise an weitere Knoten weitergereicht. Datenströme *fließen* also durch eine Reihe von Knoten (siehe Abbildung 9). Dieser Zusammenschluss von Knoten und ihren Beziehungen untereinander wird **Knotennetzwerk** genannt.

Knoten existieren jedoch nicht für sich allein, sondern sind eingebettet in eine Laufzeitumgebung, die sie startet, initialisiert und verwaltet. Diese Laufzeitumgebung wird **Knotencontainer** genannt. Dieser Knotencontainer enthält eine Menge von Knoten, aber nicht zwangsläufig alle Knoten eines Knotennetzwerkes. Die Knoten können verteilt in mehreren Knotencontainern liegen (siehe Abbildung 10). Dabei können diese Knotencontainer auch auf unterschiedlichen physikalischen Maschinen lau-

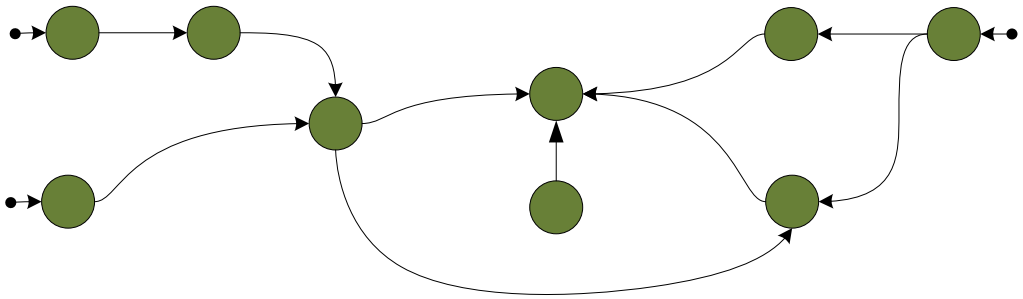


Abbildung 9: Aus drei Quellen (schwarze Punkte) geht ein Datenstrom durch eine Menge von Knoten (grüne Kreise). Knoten können dabei von mehreren Knoten einen Datenstrom empfangen, wie auch an mehrere Knoten weiterleiten

fen. Notwendig ist nur, dass ein Container auf einer Maschine läuft. Das Ziel dieser Aufteilung ist auch verteilte Datenströme abbilden zu können. Bei Datenströmen ist es oft notwendig möglichst viele Verarbeitungsschritte lokal zu erledigen und dann z.B. zur Aggregation zu einem weiteren Container zu schicken. Entfernte Kommunikation ist dabei teuer. Ein Knotencontainer bildet also eine lokale Einheit von Knoten.

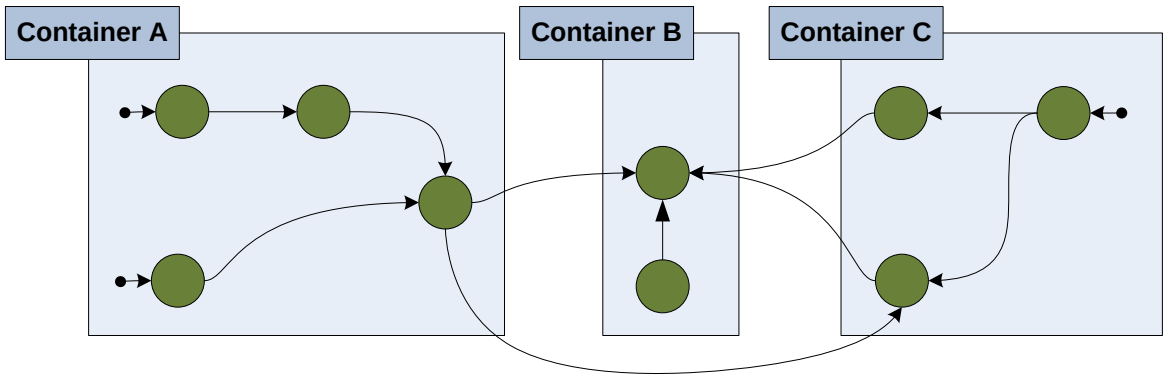


Abbildung 10: Knoten laufen auf verschiedenen Knotencontainern

Damit Knoten auch entfernt miteinander kommunizieren können, muss der Container diese entfernten Knoten finden. Dies übernimmt ein sogenannter **Namensdienst**, der die eindeutigen Namen der Knoten auflöst, falls sie lokal nicht gefunden werden konnten (siehe Abbildung 11). Jeder Container besitzt so einen Namensdienst und kann optional noch einen übergeordneten Namensdienst eines weiteren Containers kennen, den er anfragt, falls er lokal einen Knoten nicht auflösen kann. Dadurch kann man eine baumartige Vernetzung von Containern erreichen, in der spätestens der oberste Container einen Knoten auflösen kann, falls er im Netzwerk existiert.

Bei der bisher erwähnten **Kommunikation** sprechen wir von **direkten Verbindungen** zwischen Knoten. Diese Verbindungen werden statisch bei der Erstellung und

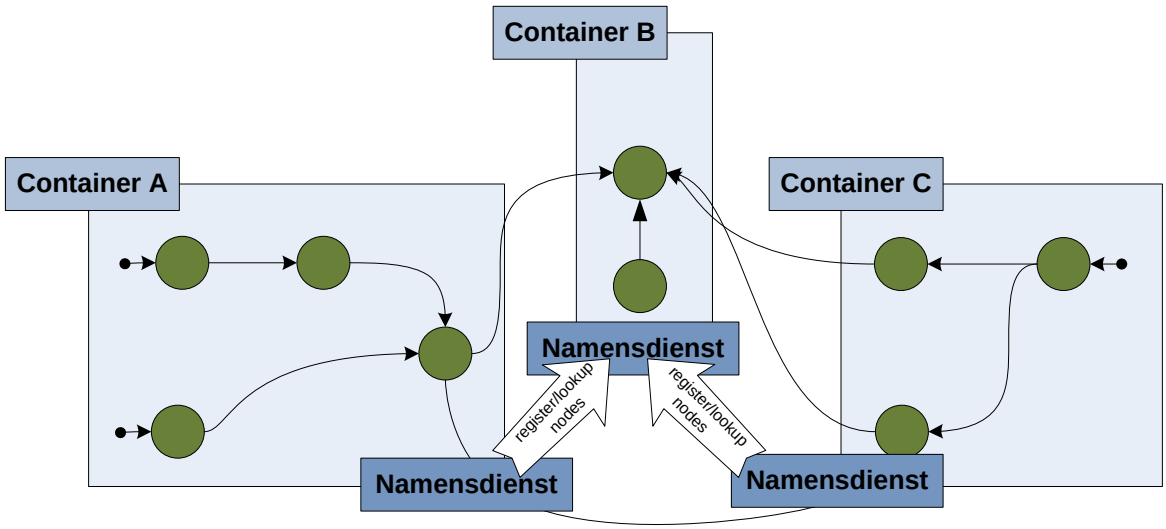


Abbildung 11: Die Knotencontainer fragen mit Hilfe eines übergeordneten Containers verteilte Knoten im Netzwerk an, damit seine lokalen Knoten mit den entfernten Knoten kommunizieren können.

Planung des Netzwerks festgelegt. Manche Knoten oder Algorithmen verlangen jedoch auch sich dynamisch vernetzen zu können, wenn sie z.B. einen bestimmten Zustand erreichen. Um dies zu erreichen kann sich ein Knoten zur Laufzeit zu einem **Themenkanal** anmelden und ein Event in einem Themenkanal veröffentlichen. Alle Knoten im gesamten Knotennetzwerk, die sich zu einem Thema angemeldet haben, erhalten dann dieses Event.

Getreu dem schon erwähnten Credo, dass entfernte Kommunikation teuer ist, sollte die Kommunikation über Themenkanäle die Ausnahme bleiben, da bei jeder Veröffentlichung das gesamte Netz nach Interessenten zu diesem Thema abgefragt wird. Es ist also empfehlenswert ein Netzwerk zu erstellen, dass die meiste Kommunikation lokal mit direkten Verbindungen vollführt.

Zusammenfassung Bei der Erstellung eines **Stream-Mining** Algorithmus zur Verarbeitung von unendlichen Datenströmen wird der Algorithmus in mehrere **Knoten** aufgeteilt. Diese Knoten laufen in **Knotencontainern**, die diese Knoten ausführen und verwalten. Mehrere Knotencontainer können entfernt zusammengeschlossen werden, so dass auch entfernte Knoten mit Hilfe des **Namensdienstes** kommunizieren können. Diese Kommunikationswege werden entweder direkt am Anfang festgelegt oder dynamisch von den Knoten zur Laufzeit durch **Themenkanäle** bestimmt. Die Gesamtheit aller Knoten auf allen Containern, die miteinander verbunden sind, wird **Knotennetzwerk** genannt.

4.2 Anwendung

Die Benutzung des Frameworks besteht aus zwei Schritten. Zunächst muss aus den Knoten ein Knotennetzwerk zusammengestellt werden. Das bedeutet, dass der Benutzer vorgeben muss, welche Knoten verwendet und wie sie untereinander verbunden werden sollen. Ist so ein Netzwerk erstellt, muss dieses dann im nächsten Schritt auch gestartet werden.

Für beide Schritte gibt es mehrere Wege, die hier in diesem Kapitel vorgestellt werden.

4.2.1 Konfiguration mit XML

XML ist eine weit verbreitete Auszeichnungssprache, die häufig zur Konfiguration von Systemen dient. Auch im Framework kann ein Knotennetzwerk mit Hilfe einer XML-Datei konfiguriert werden.

Wie schon in Kapitel 4.1 erwähnt besteht ein Knotennetzwerk aus mehreren zusammenhängenden Knoten, die den Stream-Mining Algorithmus bilden. Alle diese Knoten werden in einer XML-Datei abgebildet - unabhängig davon, ob sie jetzt auf einem physikalischen Rechner oder verteilt laufen sollen. Die physikalische Unterteilung wird dadurch definiert, dass wir den Knoten Container zuweisen. Das bedeutet für die XML-Datei, dass wir ein Netzwerk-Element mit einer Menge von Container Elementen, die wiederum Knoten-Elemente beinhalten, erstellen (siehe Abbildung 12).

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <nodenetwork xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:noNamespaceSchemaLocation="schema.xsd">
3    ...
4    <container>
5      <node>...</node>
6      <node>...</node>
7      ...
8    </container>
9
10   <container>
11     <node>...</node>
12     ...
13   </container>
14   ...
15 </nodenetwork>
```

Abbildung 12: Vereinfachte Struktur eines XML-Dokuments zur Konfiguration eines Knotennetzwerkes

Dies bildet ein Grundgerüst für unsere Konfiguration. Für die weitere Spezifikation müssen wir den Containern jetzt noch eindeutige Ids und IP-Adressen hinzufügen. Die Ids werden benötigt, um dem Framework zu sagen, welchen Container er

aus der XML laden und starten soll. Die IP-Adresse wird dazu verwendet, damit die Container unter sich kommunizieren können. Wenn ein Knotennetzwerk aus einem einzelnen Container bestehen soll, reicht es bei der IP-Adresse `localhost` anzugeben. Diese Informationen werden in einem `assign`-Tag gespeichert (siehe Abbildung 13).

```
<nodenetwork ...>
  ...
  <assign id="container-1" adress="192.168.1.1" />
  <assign id="container-2" adress="192.168.1.2" />
  ...
  <container id="container-1">...</container>
  <container id="container-2">...</container>
  ...
</nodenetwork>
```

Abbildung 13: assign-Tag zur Angabe von Container-Id und IP-Adresse

Wir haben jetzt die Container und ihre Adressen definiert. Die Instanz, die sich darum kümmert, dass die Container kommunizieren können, ist der Namensdienst. Das Framework kann hier unterschiedliche Namensdienste verwenden. Derzeit gibt es zwei Implementierungen. Zum einen einen lokalen Namensdienst, der verwendet werden kann, wenn es sich nur um einen einzelnen Container handelt und einen verteilten Namensdienst, der über Java RMI kommuniziert. Welcher Namensdienst benutzt werden soll, muss auch angegeben werden. Jeder Namensdient muss über eine `NamingServiceFactory` verfügen, die den Namensdienst instanziiert. Diese Klasse muss im `namingservicefactory` Element stehen (siehe Abbildung 14).

```
<nodenetwork ...>
  ...
  <namingservicefactory class="edu.udo.cs.pg542.naming.NamingServiceFactory"/>
  ...
</nodenetwork>
```

```
<nodenetwork>
  ...
  <namingservicefactory class="edu.udo.cs.pg542.naming.rmi.RmiNamingServiceFactory"/>
  ...
</nodenetwork>
```

Abbildung 14: Definition des zu verwendeten Namensdienstes - einmal als lokaler Namensdienst und einmal als verteilter auf RMI basierender Namensdienst.

Als nächstes wollen wir spezifizieren welche Knoten in welchem Container instanziiert werden sollen. Unterhalb des `container` Elements können beliebig vielen `node` Elemente erstellt werden. Die wichtigsten Informationen in diesem `node` Element sind zum einen die Klasse des Knoten, als auch die Id, die im ganzen Knotennetzwerk eindeutig sein muss (siehe Abbildung 15).

```
<nodenetwork ...>
  ...
  <container id="container-1">
    <node class="com.company.MyNode" id="my-node-1" />
    <node class="com.company.MyNode" id="my-node-2" />
  </container>
</nodenetwork>
```

Abbildung 15: Angabe der Klasse und Id eines Knotens

Das Framework erstellt hier nun beispielsweise einen Knoten, der im Klassenpfad unter `com.company.MyNode` instanziierbar sein muss und weist diesem die eindeutige Id `my-node-1` zu.

Wollen wir jetzt mehrere Knoten verbinden, so dass die Ausgabe von `my-node-1` zur Eingabe von `my-node-2` weitergeleitet wird, müssen wir dies auch noch angeben. In dem Knoten, der seine Ausgabe weiterreichen soll, erstellen wir ein `link` Element, das eine Liste von Zielknoten beinhaltet (siehe Abbildung 16).

Zu beachten ist hierbei, dass die Ausgabe und Eingabe von zwei Knoten kompatibel sind. Welche Ausgaben geliefert werden und welche Eingaben ein Knoten zwingend erfordert steht entweder in der Dokumentation des Knotens oder in der JavaDoc.

```
<nodenetwork ...>
  ...
  <container id="container-1">
    <node class="com.company.MyNode" id="my-node-1" >
      <link>
        <target-id>my-node-2</target-id>
        <target-id>...</target-id>
      </link>
    </node>
    <node class="com.company.MyNode" id="my-node-2" />
  </container>
  ...
</nodenetwork>
```

Abbildung 16: Angabe der Klasse und Id eines Knotens

Die angegebenen Ids der Ziele müssen im Knotennetzwerk vorhanden sein - entweder lokal auf dem gleichen Container oder auf einem anderen Container. Das Framework wird dann erst einmal lokal versuchen den Zielknoten zu finden und ansonsten seinen übergeordneten Container fragen. Dafür muss jedoch noch angegeben werden welcher Container einem anderen Container übergeordnet ist. Dies kann man einfach im Attribut `parentId` vom `container` Element definieren - wobei natürlich auch hier ein Container mit dieser Id existieren muss (siehe Abbildung 17).

Hat man so die Container verbunden, dann kann ein Knoten aus `container-1` beispielsweise mit einem Knoten aus `container-2` verbunden werden, wobei der Container `master` die Namensauflösung übernimmt, da er von beiden Containern der übergeordnete Container ist.

```
<nodenetwork ...>
  ...
  <container id="container-1" parentId="master" >...</container>
  <container id="container-2" parentId="master" >...</container>
  <container id="master">...</container>
  ...
</nodenetwork>
```

Abbildung 17: Verbindung der Container untereinander

Parameter Die wichtigsten Elemente eines Knotennetzwerkes haben wir nun fast alle gesehen. Ein Netzwerk und seine Knoten würden derzeit aber relativ unflexibel sein, wenn man sie nicht parametrisieren könnte. Das Framework bietet dabei zwei Arten von Parametern. Globale Parameter werden außerhalb eines `container` Elements definiert und können von allen Knoten ausgelesen werden. Knotenparameter werden dagegen innerhalb eines `node` Elements angegeben und können auch nur von diesem Knoten aus gelesen werden. Angegeben werden bei beiden Formen jeweils der Name des Parameters, unter dem der Knoten den Parameter auslesen kann, dann der Datentyp des Parameters und schließlich sein Wert.

```
<nodenetwork ...>
  ...
  <parameter name="global-single-par" type="Long" value="1000" />
  <parameter name="global-list-par" type="LongList" value="1000;2000;3000" />
  ...
  <container id="container-1">
    <node class="com.company.MyNode" id="my-node-1">
      <parameter name="single-par" type="Long" value="1000" />
      <parameter name="list-par" type="LongList" value="1000;2000;3000" />
    </node>
  </container>
  ...
</nodenetwork>
```

Abbildung 18: Definition von globalen, wie auch Knotenparametern

Die möglichen Datentypen sind dabei Folgende:

- String
- Integer
- Boolean
- Double
- Float
- Long

Zusätzlich kann man an den Datentyp noch `List` anfügen, als z.B. `StringList` und kann eine Liste dieses Datentyps erstellen, dessen Werte durch ein Semikolon getrennt werden.

Welcher Knoten nun welchen Datentyp braucht, oder welcher Parameter zwingend oder optional benötigt wird, kann man am besten in der Dokumentation/JavaDoc des jeweiligen Knoten nachschlagen. Für weitere Informationen siehe Kapitel 4.3.4.

Kickoff Nun muss nur noch definiert werden, welche Knoten initial mit einem leeren Event (also ohne Daten) gestartet werde sollen. Dies sind die sogenannten Kickoff-Knoten. Jeder Knoten, der beim Start eines Containers gestartet werden soll, muss ein Attribut `kickoff` mit dem Wert `true` gesetzt haben (siehe auch in Abbildung 19).

Version Abschließend muss noch eine Version des Knotennetzwerkes auf oberster Ebene anhand des Elements `version` angegeben werden und wir können dieses Netzwerk dann schließlich starten (siehe Kapitel 4.2.2). Es gibt zwar noch die Möglichkeit durch Angabe von sogenannte Hooks in der XML, die Funktionalität des Frameworks für gewisse Anwendungen (siehe Kapitel 5 über die Evaluation) zu erweitern, aber dies soll nicht in diesem Kapitel behandelt werden.

In Abbildung 19 ist nun eine komplette, beispielhafte XML zur Knotennetzwerkkonfiguration dargestellt.

4.2.2 Start einer XML-Konfiguration über die Konsole

Um das erstellte Knotennetzwerk nun zu starten, brauchen wir zunächst eine lauffähige Version des Frameworks. Wir beschreiben das Vorgehen anhand von Maven

Es gibt drei Maven-Projekte, die in der Projektgruppe entstanden sind, aus denen man eine Jar-Datei erstellen kann.

pg542-core Der pure Kern des Frameworks ohne konkrete Knotenimplementierungen, Algorithmen oder Hilfsklassen.

pg542-util Eine Ansammlung von verschiedenen Knoten- und Algorithmenimplementierungen. Bindet `pg542-core` automatisch ein.

pg542-wads Der Anwendungsfall aus der Projektgruppe. Bindet `pg542-core` und `pg542-util` automatisch ein.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<nodenetwork xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="schema.xsd">
  <version>0.22</version>

  <namingservicefactory class="edu.udo.cs.pg542.naming.rmi.RmiNamingServiceFactory"/>

  <assign id="container-1" adress="192.168.1.1" />
  <assign id="container-2" adress="192.168.1.2" />
  <assign id="master" adress="192.168.1.3" />

  <parameter name="global-1" type="Boolean" value="true" />
  <parameter name="global-2" type="StringList" value="v1;v2" />

  <container id="container-1" parentId="master" >
    <node class="com.company.MyNode" id="my-node-1" kickoff="true">
      <parameter name="single-par" type="Long" value="1000" />
      <link>
        <target-id>my-node-2</target-id>
      </link>
    </node>
  </container>

  <container id="container-2" parentId="master" >
    <node class="com.company.MyNode" id="my-node-2" />
  </container>

  <container id="master" />

</nodenetwork>
```

Abbildung 19: Vollständige Beispiel XML-Konfiguration

Möchte man ein eigenes Maven-Projekt erstellen, muss man mindestens `pg542-core` (`pg542-util` ist optional und stellt verschiedene Hilfsklassen und Knoten zu Verfügung) als Abhängigkeit hinzufügen. Zwecks einfacheren Kompilierens und Startens sollte zusätzlich noch das `assembly`-Plugin aktiviert und konfiguriert (siehe Abbildung 20) werden.

Mit Hilfe des `assembly`-Plugins können wir nun schnell und einfach das Projekt kompilieren und starten. Nachdem wir `mvn assembly:assembly` in die Konsole (im Verzeichnis der `pom.xml`) eingeben, werden zwei Jar-Dateien im `target`-Ordner erstellt:

<name>-<version>.jar In dieser Jar-Datei sind nur die kompilierten Klassen aus dem eigenen Projekt. Diese Datei alleine ist nicht lauffähig, wenn die benötigten Abhängigkeiten nicht im Klassenpfad sind.

<name>-<version>-jar-with-dependencies.jar In dieser Datei wurden auch alle Abhängigkeiten mitkopiert. D.h. das diese Version zwar deutlich größer, dafür brauchen wir keine weiteren Klassen im Klassenpfad. Wir gehen im Weiteren von dieser Datei aus.

Nun haben wir ein Knotennetzwerk in einer XML erstellt und alle benötigten Klassen in einer lauffähigen Jar-Datei. Wenn beides in einem Ordner liegt, dann genügt

```xml
<project ...>
  ...
  <repositories>
    <repository>
      <id>pg542</id>
      <name>pg542</name>
      <url>http://maven-ai.cs.uni-dortmund.de/repository/pg542</url>
    </repository>
  </repositories>
  <dependencies>
    ...
    <dependency>
      <groupId>edu.udo.cs.pg542</groupId>
      <artifactId>pg542-core</artifactId>
      <version>0.2.2-SNAPSHOT</version>
      <type>jar</type>
      <scope>compile</scope>
    </dependency>
    <dependency>
      <groupId>edu.udo.cs.pg542</groupId>
      <artifactId>pg542-util</artifactId>
      <version>0.1.1-SNAPSHOT</version>
      <type>jar</type>
      <scope>compile</scope>
    </dependency>
  </dependencies>
  ...
  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-jar-plugin</artifactId>
        <configuration>
          <archive>
            <manifest>
              <addClasspath>true</addClasspath>
              <mainClass>edu.udo.cs.pg542.core.ui.Console</mainClass>
            </manifest>
          </archive>
        </configuration>
      </plugin>
      <plugin>
        <artifactId>maven-assembly-plugin</artifactId>
        <configuration>
          <descriptorRefs>
            <descriptorRef>jar-with-dependencies</descriptorRef>
          </descriptorRefs>
          <archive>
            <manifest>
              <addClasspath>true</addClasspath>
              <mainClass>edu.udo.cs.pg542.core.ui.Console</mainClass>
            </manifest>
          </archive>
        </configuration>
      </plugin>
      ...
    </plugins>
  </build>
  ...
</project>
```

Abbildung 20: Einbinden des Frameworks in ein eigenes Maven-Projekt (pom.xml)

folgender Befehl, um das Netzwerk aus einer XML mit vorgegebener Container-Id zu booten.

```
java -jar <name>-<version>-jar-with-dependencies.jar -xml <xml-file> -id <container-id>
```

Wenn das Netzwerk gebootet ist, wurden alle Knoten korrekt instanziiert und initiiert. Die `kickoff`-Knoten wurden aber noch nicht gestartet. Das Framework verlangt nach dem Booten eine Bestätigung zum Starten der `kickoff`-Knoten. Dies ist wichtig, da beim Starten eines Knoten alle entfernten Verbindungen aufgelöst werden. Das bedeutet insbesondere, dass alle benötigten Container gebootet sein müssen. Deswegen bootet man zunächst alle Container und startet sie, sobald alle fertig sind.

4.2.3 Konfiguration und Start mit Java

Eine weitere Möglichkeit das Framework zu benutzen ist, es von seinem eigenen Java-Programm aus zu konfigurieren und zu starten. Dies kann insbesondere beim Testen und Debuggen sehr hilfreich sein. Notwendig hierfür ist natürlich wieder, dass die Jar-Dateien des Frameworks (`pg542-core` und optional `pg542-util`) im Klassenpfad der Anwendung liegen.

Netze erstellen Bisher haben wir ein Knotennetzwerk nur über eine XML-Datei erstellt und konfiguriert. Es gibt aber auch die Möglichkeit dies in Java zu tun. Hierfür gibt es das Interface *NodeNetwork*. Jede Implementierung dieses Interfaces beinhaltet alle Informationen, die notwendig sind, um ein Teilnetz innerhalb eines Knotencontainers zu erstellen. Die Klasse *XmlNodeNetwork* implementiert das Interface indem es eine XML-Datei liest, parst, die Elemente zu seiner zugewiesenen Container-Id sucht und die jeweiligen Informationen über die *NodeNetwork*-Schnittstelle zurückgibt. Es reicht also aus, sich mit dem Interface vertraut zu machen und dahingehend ein Knotennetzwerk in Java zu programmieren.

Das Framework bietet neben der Implementierung *XmlNodeNetwork* auch noch *NodeNetworkPojo* an. Diese Klasse bietet eine Reihe von *set*-Methoden an, die die Netzwerkinformationen speichern und über das Interface anbieten. Dies bietet die Möglichkeit ein Netzwerk schnell innerhalb einer Methode zu erstellen und zu starten, ohne eine Klasse zu implementieren. Abbildung 21 demonstriert dieses Vorgehen.

Will man mehrere verteilte Container starten, dann erstellt man für jeden Container so ein Netzwerk, achtet darauf eindeutige Ids zu verteilen und erreichbare IP-Adressen einzustellen. Wie bei der XML-Datei müssen die Container dann nur noch untereinander verbunden werden, indem manchen Containern übergeordnete Container angegeben werden (siehe Abbildung 22).

```java
NodeNetworkPojo net = new NodeNetworkPojo();

// Setze Container-Id und Addresse
net.setContainerAddress("localhost");
net.setContainerId("container");

// Lokalen Namensdienst setzen
net.setNamingServiceFactoryClass(NamingServiceFactory.class);

// Knoten hinzufügen
net.addNodeClass("node-1", MyInputNode.class);
net.addNodeClass("node-2", MyOutputNode.class);

// Parameter für Knoten hinzufügen
net.addParameter("node-1", "file", "data.csv");
net.addParameter("node-2", "level", "severity");

// Knoten verbinden
net.addFollower("node-1", "node-2");

// Startknoten definieren
net.addKickoff("node-1");
```

Abbildung 21: Konfiguration eines lokalen Knotennetzwerks über NodeNetworkPojo

```java
// Auf Rechner A
NodeNetworkPojo netA = new NodeNetworkPojo();

netA.setContainerAddress("192.168.1.1");
netA.setContainerId("container-1");

// RMI Namensdienst setzen
netA.setNamingServiceFactoryClass(RmiNamingServiceFactory.class);

// Netz für Container A erstellen, inkl. Verbindungen zu Knoten in Container B
...

// Auf Rechner B
NodeNetworkPojo netB = new NodeNetworkPojo();

netB.setContainerAddress("192.168.1.2");
netB.setContainerId("container-2");

// RMI Namesdienst setzen
netB.setNamingServiceFactoryClass(RmiNamingServiceFactory.class);
netB.setParentContainerAddress("192.168.1.1");
netB.setParentContainerId("container-1");

// Netz für Container B erstellen, inkl. Verbindungen zu Knoten in Container A
...
```

Abbildung 22: Konfiguration eines verteilten Knotennetzwerkes über NodeNetwork-Pojos

Netze starten Unabhängig davon, ob wir die Schnittstelle *NodeNetwork* nun selbst implementiert haben, oder die Klasse *NodeNetworkPojo* benutzen, oder einfach nur *XmlNodeNetwork* manuell einsetzen, haben wir jetzt ein *NodeNetwork*-Instanz, die ein

(Teil eines) Netzwerk darstellt. Dies wollen wir jetzt booten.

Das Booten eines Container besteht dabei aus vielen einzelnen Schritten. Der Container muss erstellt und zusammen mit dem Namensdienst initialisiert werden, Knoten müssen instanziiert und initialisiert werden, Parameter injiziert, usw. All diese vielen kleinen Aufgaben übernimmt eine Klasse namens *ContainerBootstrap*. Beim Erstellen einer Instanz wird dieser Klasse eine *NodeNetwork*-Instanz übergeben. Beim Aufruf der Methode *boot()* wird das Knotennetzwerk ausgelesen, ein Knotencontainer inklusive all seiner Knoten erstellt und zurückgegeben. Die Knoten innerhalb dieses Containers sind nun voll funktionstüchtig - können also schon entfernte Aufrufe von anderen Knoten empfangen. Nur die sogenannten `kickoff`-Knoten wurden noch nicht gestartet, damit erst einmal das ganze verteilte Netz booten kann, bevor der erste Knoten anfängt.

Wenn aber alle Container bereit sind, dann kann die Methode *run(NodeContainer)* aufgerufen werden, die dann alle Startknoten anstößt (siehe Abbildung 23).

Weitere Informationen zum Bootvorgang befinden sich in Kapitel 4.4.4.

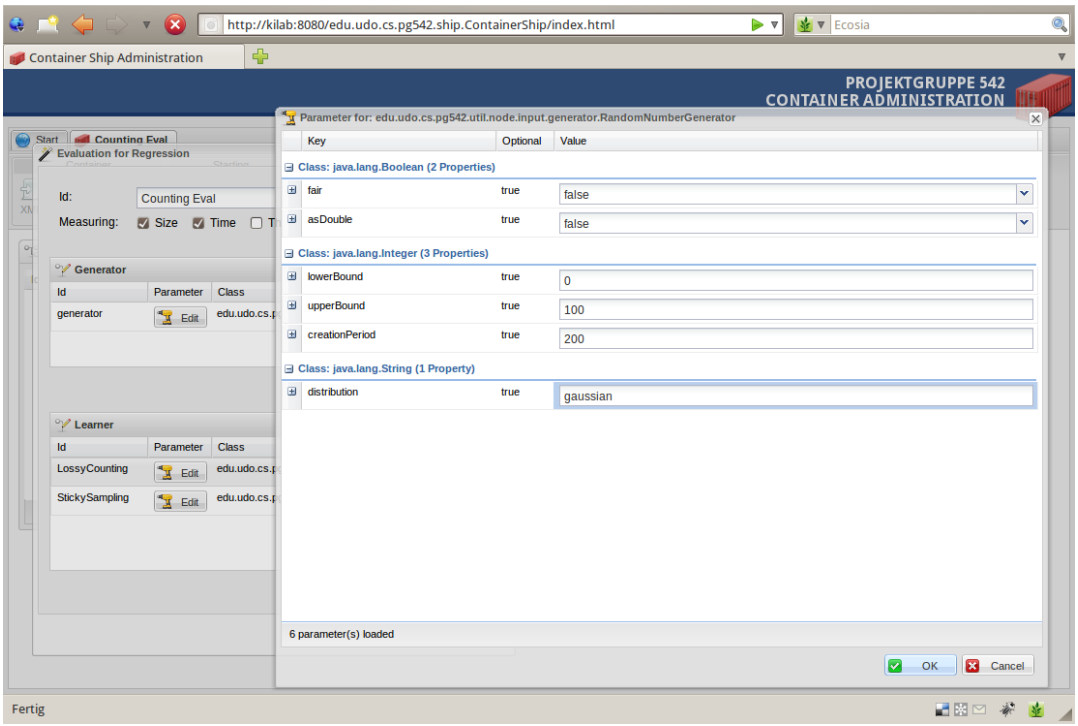
```
1  NodeNetworkPojo net = new NodeNetworkPojo();
2
3  // Konfiguriere das Knotennetzwerk
4  ...
5
6  ContainerBootstrap boot = new ContainerBootstrap(net);
7  NodeContainer container = boot.boot();
8
9  // Warte auf andere Container im Netzwerk
10 boot.run(container);
```

Abbildung 23: Booten und Starten eines Knotennetzwerkes

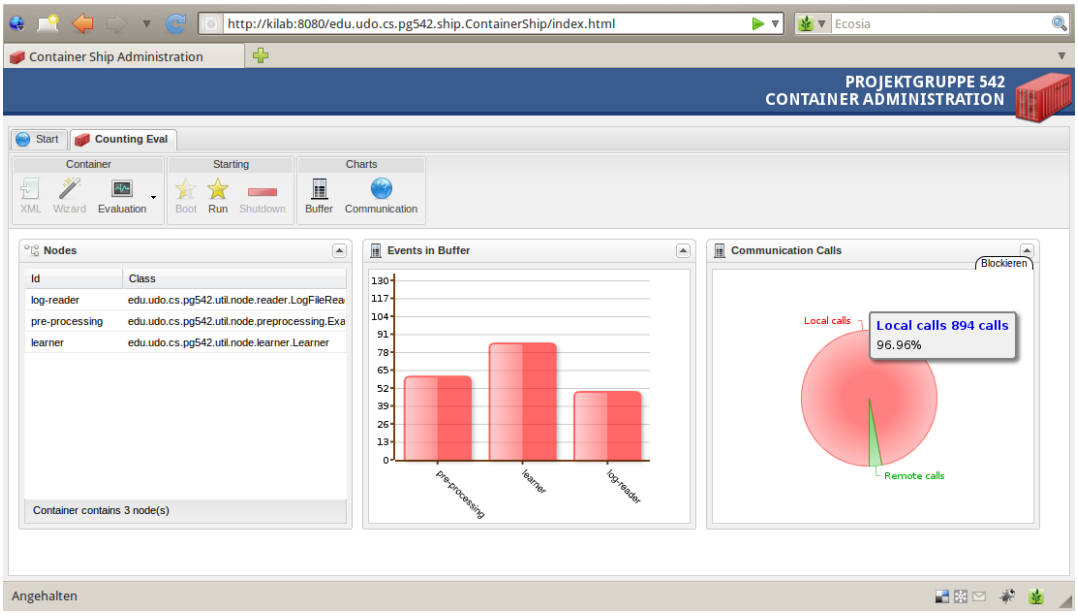
4.2.4 Webanwendung

Das Projekte `pg542-ship` bietet die Möglichkeit Netzwerke und Container über eine Web-Schnittstelle zu verwalten. Beim Starten der `jar`-Datei wird automatisch ein Webserver gestartet und eine Webanwendung bereitgestellt, über die man Netzwerke erstellen, Container booten und starten kann (siehe Abbildung 24a).

Nützlich ist dabei nicht nur die Verwaltung von Containern, sondern auch das Überwachen von Speicherbedarf, Buffern und Kommunikation (siehe Abbildung 24b). Auch das Anzeigen von Diagrammen bei der Evaluation vereinfacht das Vergleichen von verschiedenen Streamalgorithmen.



(a) Erstellung von Knoten



(b) Überwachung von Containern

Abbildung 24: Webanwendung zur Administration von Knotencontainern

4.3 Knotenentwicklung

Dieses Kapitel widmet sich der Entwicklung von eigenen Knoten, Lernalgorithmen und -modellen. Zunächst wird im folgendem Kapitel an einem kleinen Beispiel die Entwicklung eines simplen Knoten erklärt. Das Ziel dieses Kapitels ist es möglichst schnell an die Programmierung eines Knotens heranzuführen. Dabei wird nicht auf Details eingegangen sondern auf die jeweiligen folgenden Kapitel verwiesen, die sich dieser Thematik jeweils detaillierter widmen.

4.3.1 Knoten

Zur Veranschaulichung der Knotenentwicklung werden wir einen simplen Knoten programmieren, der beliebige Strings empfängt und bei einem bestimmten Signal die k-häufigsten Strings an den nächsten Knoten weiterschickt.

Die schnellste und einfachste Möglichkeit diesen Knoten zu implementieren, ist das Erben von der Klasse *AbstractNode*. In dieser Klasse müssen wir zwei Methoden implementieren und können durch Überschreiben von weiteren, schon vorhandenen, aber leer implementierten Methoden, zusätzliche Funktionalität hinzufügen.

Im Leben eines Knoten gibt es zwei Phasen. Die Initialisierungs- und Ausführungsphase. Für jede dieser Phasen gibt es zuständige Methoden in denen der Knoten implementiert wird.

Initialisierungsphase Beim Booten eines Containers wird jeder Knoten im Netzwerk initialisiert. Er wird in seine Umgebung eingefügt und soll in dieser Initialisierungsphase Informationen über sich an den Container weiter geben.

Die Methode, die zuerst aufgerufen wird, leitet Informationen zu den benötigten Parametern an den Container, damit dieser die Parameter validieren und in den Knoten injizieren kann. Dies passiert in der Methode *addParameterTypes*, die in *AbstractNode* leer implementiert ist und optional überschrieben werden kann. Diese Methode bekommt eine *Collection* zu der jeder Knoten seine benötigten Parametertypen hinzufügt. Nähere Details zu Parametertypen befinden sich im Kapitel 4.3.4. Wichtig für uns ist, dass wir für die k-häufigsten Zeichenketten per Parameter das k angeben lassen wollen (siehe Programmcode in Abbildung 25).

Der nächste Schritt ist der Aufruf der *init*-Methode. Diese Methode ist beispielsweise gut dafür geeignet Parameterwerte abzurufen. Der Knotencontainer hat alle Parameter nun validiert und stellt sie dem Knoten zur Verfügung. In unserem Beispiel holen wir das k und instanziieren den eigentlichen Lerner, der als Modell die Top-K Zeichenketten zurückgibt (siehe Programmcode in Abbildung 26, Zeile 18-19). Wie genau Lerner und Modelle zu implementieren sind, steht im Kapitel 4.3.3. Wichtig

```java
public class TopKNode extends AbstractNode {

  /**
   * Schlüssel für den Parameter k. Schlüssel werden oft
   * als Konstante notiert und mit einem Präfix versehen.
   * PK_ steht dabei für "ParameterKey", EK_ für "EventKey"
   * und  TK_ für "TopicKey"
   */
  private static final String PK_K = "k"

  @Override
  public void addParameterTypes(Collection<ParameterType<?>> parameterTypes) {
    IntegerParameterType kType = ParameterType.getInteger(PK_K);
    // Nur positive Werte für k
    kType.setLowerBound(0);
    interval.setDescription("Determines how many frequent strings will be returned");

    parameterTypes.add(interval);
  }
  ...
}
```

Abbildung 25: addParameter-Methode für den Top-K Knoten

für uns ist erst einmal das dieser Lerner auf magische Art und Weise die k-häufigsten Strings auf einem Datenstrom sucht.

```java
public class TopKNode extends AbstractNode {

  /**
   * Schlüssel für den Parameter k. Schlüssel werden oft
   * als Konstante notiert und mit einem Präfix versehen.
   * PK_ steht dabei für "ParameterKey", EK_ für "EventKey"
   * und  TK_ für "TopicKey"
   */
  private static final String PK_K = "k"
  private static final String EK_INPUT = "string-input"
  private static final String EK_TOP_K = "top-k"

  private Learner<String, DescriptionModel<String>> learner;
  ...

  @Override
  protected void init() {

    int k = (Integer) getParameter(PK_K);
    learner = new TopKLearner<String>(k);

    requiresInput(EK_INPUT, String.class);
    assuresOutput(EK_TOP_K, DescriptionModel.class);
  }
}
```

Abbildung 26: addParameter-Methode für den Top-K Knoten

Ein weiterer wichtiger Punkt in der Initialisierungsphase ist das Angeben von Ein- und Ausgaben. Knoten können prinzipiell von überall Daten vom unterschiedlichen Typ bekommen. Typsicherheit zur Laufzeit ist da ein Problem. Um dieses Problem zu

mindern, kann ein Knoten angeben, welche Eingabe er auf jeden Fall benötigt und wenn er eine Ausgabe verschickt, welchen Typ er zusichert. Dies sind keine Angaben die das Framework zur Laufzeit überprüft, sondern freiwillige Angaben an die der Knoten sich halten muss. Das Framework überprüft nur einmal beim Booten des Netzwerkes, ob zwei Knoten die verbunden sind, bei der Ein- und Ausgabe kompatibel sind.

Bei unserem Beispiel wollen wir Zeichenketten zählen, also erwarten wir auch unter einem bestimmten Schlüssel im eingehenden *Event* (für weitere Informationen zur Kommunikation mit *Event*s siehe Kapitel 4.3.2) ein Objekt vom Typ *String*. Wenn wir eine Ausgabe erzeugen, dann sichern wir zu, dass wir ein Top-K-Modell verschicken (siehe Programmcode in Abbildung 26, Zeile 21-22).

Unter welchen Schlüssel die Daten liegt, ist hier fest über Konstanten definiert. Um den Knoten generischer zu gestalten, könnte man den Schlüssel auch über einen Parameter definieren. In seiner eigenen Knotenbibliothek ist es auch denkbar ein Interface zu erstellen, das als Sammlung von Schlüsseln fungiert und an dem sich alle Klassen bedienen.

Ausführungsphase Nach der Initialisierung eines Knotens beginnt die eigentliche Arbeit. Hier gibt es zwei Methoden, die eingehende *Event*s verarbeiten. Die Wichtigste ist dabei die *execute*-Methode. Diese Methode wird mit dem nächsten *Event*, das in der Warteschlange zwischengespeichert ist, aufgerufen. Von da an entscheidet der Knoten, was mit diesem *Event* passiert.

Unser Top-K-Knoten hat hier eine sehr einfache Aufgabe. Er extrahiert den zu zählenden String aus dem *Event* und überreicht ihn unserer Lerner-Instanz (siehe Programmcode in Abbildung 27, Zeile 17-18). Das *Event* wird nur auf Anfrage weitergeschickt. Diese Anfrage erwarten wir jedoch nicht als `DATA`- sondern als `SIGNAL`-*Event* (siehe Kapitel 4.3.2). Signale werden bevorzugt verschickt und werden nicht an die *execute*-Methode gereicht, sondern an die Methode *receivedSignal*. Diese Methode ist in *AbstractNode* leer implementiert und kann überschrieben werden. Wir werden auf so ein Signal mit der Versendung unseres Top-K-Modells über die *dispatch*-Methode reagieren (siehe Programmcode in Abbildung 27, Zeile 23-24).

Somit wäre unser Knoten schon einsatzbereit. Er verarbeitet Zeichenketten indem er ihn an einen Top-K-Lerner weiter leitet und reagiert auf ein Signal indem er das Modell weiterschickt. Wir implementieren aber noch eine weitere Funktionalität, um den Einsatz von Signalen und Themenkanälen zu demonstrieren.

Um andere Knoten im Netzwerk darüber zu informieren, wie viele Zeichenketten unser Knoten schon verarbeitet hat, verschicken wir ein Signal über ein Themenkanal. Wenn ein anderer Knoten sich entscheidet, dass genug Strings verarbeitet wurden, dann verschickt es wiederum ein Signal, damit unser Knoten sein Ergebnis verschickt. Wir werden so ein Signal der Einfachheit halber alle 1000 Strings verschi-

```
1  public class TopKNode extends AbstractNode {
2
3    /**
4     * Schlüssel für den Parameter k. Schlüssel werden oft
5     * als Konstante notiert und mit einem Präfix versehen.
6     * PK_ steht dabei für "ParameterKey", EK_ für "EventKey"
7     * und  TK_ für "TopicKey"
8     */
9    private static final String PK_K = "k"
10   private static final String EK_INPUT = "string-input"
11   private static final String EK_TOP_K = "top-k"
12
13   private Learner<String, DescriptionModel<String>> learner;
14   ...
15
16   @Override
17   public void execute(Event event) {
18     String string = (String)event.get(EK_INPUT);
19     learner.learn(string);
20   }
21
22   @Override
23   public void receivedSignal(Event event) {
24     // Wir löschen alle Daten im Event ...
25     event.clear();
26     // ... und setzen unser Modell
27     event.set(EK_TOP_K, learner.getModel());
28     dispatch(event);
29   }
30
31 }
```

Abbildung 27: execute- und receivedSignal-Methoden für den Top-K Knoten

cken, aber natürlich wäre es geschickter dies auch über einen Parameter steuern zu lassen.

Wir passen unsere *execute*-Methode an, indem wir für jeden Aufruf einen Zähler vergleichen und erhöhen. Haben wir genug Zeichenketten gezählt, dann leeren wir das *Event* (Vermeidung von unnötigen Overhead bei der Versendung), ändern den Typen zu einem Signal und verschicken es über ein Topic mit einem festgelegten Schlüssel. Für das Verschicken ist hier nun die Methode *publish* zuständig (siehe Abbildung 28, Zeile 27-30).

Bleibt dann nur noch die Frage, wie sich ein Knoten zu einem Themenkanal anmeldet. Da Signale in den meisten Fällen über Themenkanäle versendet werden, wollen wir auch hier keine Ausnahme machen und uns zu dem Kanal anmelden, der die Signale zur Ausgabe beinhaltet. Durch ein simplen Aufruf von *subscribe* (bzw. *unsubscribe* zum Abmelden) sind wir damit auch schon fertig. Erfolgen kann dieser Aufruf auch in der *init*-Methode. Das Ergebnis unserer Knotenimplementierung inklusiv dieser Neuerung ist in Abbildung 29 noch einmal komplett aufgelistet.

```java
public class TopKNode extends AbstractNode {

  /**
   * Schlüssel für den Parameter k. Schlüssel werden oft
   * als Konstante notiert und mit einem Präfix versehen.
   * PK_ steht dabei für "ParameterKey", EK_ für "EventKey"
   * und  TK_ für "TopicKey"
   */
  private static final String PK_K = "k"
  private static final String EK_INPUT = "string-input"
  private static final String EK_TOP_K = "top-k"

  private static final String TK_COUNT = "string-count-topic";
  private static final String EK_COUNT = "string-count";


  private Learner<String, DescriptionModel<String>> learner;

  private int counter = 0;
  ...

  @Override
  public void execute(Event event) {
    String string = (String)event.get(EK_INPUT);
    learner.learn(string);
    if( counter++>1000 ) {
      event.clear();
      event.setType(EventType.SIGNAL);
      event.set(EK_COUNT, 1000);
      publish(event, new Topic(TK_COUNT));
    }
  }
  ...
}
```

Abbildung 28: Verschicken von Signalen über einen Themenkanal

4.3.2 Kommunikation

Knoten zu entwickeln bedeutet nicht nur Daten zu verarbeiten, sondern auch diese erst einmal zu bekommen und an andere Knoten weiter zu schicken. In diesem Kapitel werden alle relevanten Themen zur Knotenentwicklung und Kommunikation erläutert.

Event Die Daten, die zwischen den Knoten ausgetauscht werden, werden in einzelnen Paketen, sogenannten *Event*s verschickt. Ein *Event* ist hierbei aber nicht nur ein Datum, sondern eine Menge von Objekten, die als Einheit zusammen zu verarbeiten sind. Diese Objekte werden von einem Knoten in einem *Event* unter verschiedenen (String-)Schlüsseln gespeichert und können unter diesen Schlüsseln auch wieder abgerufen werden. Zu beachten ist hierbei, dass diese gespeicherten Objekte auch serialisierbar sein müssen, da sie unter Umständen auch verteilt verschickt werden. Im Prinzip ist ein *Event* also eine aus Java bekannte *Map*. Zusätzlich zu dem assoziativen Speichern von Objekten, werden hier aber auch Zeitstempel oder Flags gespeichert.

```java
public class TopKNode extends AbstractNode {

  private static final String PK_K = "k"
  private static final String EK_INPUT = "string-input"
  private static final String EK_TOP_K = "top-k"
  private static final String EK_COUNT = "string-count";

  private static final String TK_OUTPUT = "output-topic";
  private static final String TK_COUNT = "string-count-topic";

  private Learner<String, DescriptionModel<String>> learner;
  private int counter = 0;

  @Override
  public void addParameterTypes(Collection<ParameterType<?>> parameterTypes) {
    IntegerParameterType kType = ParameterType.getInteger(PK_K);
    // Nur positive Werte für k
    kType.setLowerBound(0);
    interval.setDescription("Determines how many frequent strings will be returned");

    parameterTypes.add(interval);
  }

  @Override
  protected void init() {

    int k = (Integer) getParameter(PK_K);
    learner = new TopKLearner<String>(k);

    requiresInput(EK_INPUT, String.class);
    assuresOutput(EK_TOP_K, DescriptionModel.class);

    subscribe(new Topic(TK_OUTPUT))
  }

  @Override
  public void execute(Event event) {
    String string = (String)event.get(EK_INPUT);
    learner.learn(string);
    if( counter++>1000 ) {
      event.clear();
      event.setType(EventType.SIGNAL);
      event.set(EK_COUNT, 1000);
      publish(event, new Topic(TK_COUNT));
    }
  }

  @Override
  public void receivedSignal(Event event) {
    // Wir löschen alle Daten im Event ...
    event.clear();
    // ... und setzen unser Modell
    event.set(EK_TOP_K, learner.getModel());
    dispatch(event);
  }
}
```

Abbildung 29: Kompletter Top-K-Knoten

EventType Weiterhin gibt es noch eine wichtige Unterscheidung. Von einem *Event* gibt es zwei verschiedene Typen (*EventType*).

EventType.DATA Ein *Event* was zu verarbeitende Daten zwischen zwei Knoten überträgt. Dies ist der normalerweise verwendete Typ zur Kommunikation zwischen Knoten.

EventType.SIGNAL Ein Steuersignal, das für die Verwaltung der Knoten benutzt wird, um z.B. den Status eines Knoten zu ändern.

Je nachdem um welchen Typen es sich handelt, werden die *Event*s bei der Übertragung unterschiedlich behandelt. Wird ein *Event* vom Typ `DATA` an ein Knoten geschickt, während dieser noch ein anderes *Event* verarbeitet, dann wird er nach dem FIFO-Prinzip (First In First Out) in eine Warteschlange eingereiht. *Event*s vom Typ `SIGNAL` haben dagegen eine höhere Priorität und werden an den wartenden *Event*s direkt zum Knoten geleitet. Dort wird nachdem das derzeitige *Event* verarbeitet wurde, direkt eine spezielle, zur Signalverarbeitung bestimmte Methode aufgerufen. Ob der Knoten diese Methode implementiert und auf das Signal reagiert, bleibt dem Entwickler überlassen. So kann jedoch schnell und zuverlässig ein Status einzelner Knoten geändert werden.

Dispatching Es gibt zwei verschiedene Wege *Event*s zwischen Knoten auszutauschen. Der normale und schnellste Weg wird **Dispatching** genannt. Bei der Erstellung eines Knotennetzwerkes werden Knoten miteinander verbunden. Die Ausgabe eines Knoten führt zur Eingabe eines anderen Knotens. Diese Verbindung werden einmal statisch bei der Erstellung festgelegt (siehe Kapitel 4.2) und können zur Laufzeit nicht mehr geändert werden. Dieser Pfad bildet eine Art Pipeline, das ein Packet nach dem anderen bearbeitet.

Publishing Neben dem Dispatching gibt es noch eine dynamische Möglichkeit zur Kommunikation. Beim **Publishing** können sich Knoten zur Laufzeit zu sogenannten Themenkanälen (**Topic**s) an- und wieder abmelden. Jedes *Event* was an so einen Themenkanal veröffentlicht wird (Publishing), wird an alle angemeldeten Knoten verschickt. Da bei jeder Veröffentlichung nach allen angemeldeten Knoten gesucht wird, ist diese Art der Kommunikation deutlich langsamer als die statische Variante. Empfehlenswert ist es das Dispatching für die Hauptkommunikation zu benutzen und Publishing für Kommunikation außerhalb der normalen Datenpfade. Beispielsweise, um verteilte Modelle zu aktualisieren oder um den Status von einer Menge von Knoten zu ändern (z.B. mit einem `SIGNAL`-*Event*).

4.3.3 Lerner & Modell

Viele Knoten in einem Netzwerk sind zwar dazu da Eingaben zu lesen, vorzuverarbeiten oder auszugeben, aber die wichtigsten Knoten bleiben die Lernknoten. Dies

bedeutet, dass der Knoten einen Strom an Daten bekommt und auf diesem ein Modell lernt. Dabei gibt es zwei verschiedene Arten von Modellen, beschreibende und vorhersagende Modelle. Ein beschreibendes Modell stellt ein Abbild der Wirklichkeit, also des Datenstroms, zur Verfügung. Ein sehr einfaches beschreibendes Modell wäre es beispielsweise die k häufigsten Elemente auszugeben. Ein Modell, das vorhersagt, dient z.B. zur Klassifikation oder Regression von Daten.

Ein Lernknoten sollte nun diese zwei Komponenten, also einen Lerner und ein gelerntes Modell, berücksichtigen und vom eigentlichen Knoten abkapseln. Dies dient vor allem der Übersicht und der Wiederverwendbarkeit. Die eigentlichen Lernalgorithmen und Modelle sollten sich nur mit ihrer eigentlichen Aufgabe beschäftigen, ohne sich mit knoteninternen Details auseinander zu setzen. Der Knoten selbst sollte sich diesen Aufgaben widmen, d.h. Parameter angeben, Ein- und Ausgabe spezifizieren, Daten aus einem *Event* extrahieren, usw., um dann mit diesen Daten den Lerner aufzurufen, der dann die eigentliche Arbeit macht. Wird ein Ergebnis von diesem Lerner benötigt, wird nach dem Modell gefragt und dieses zur Weiterverarbeitung verschickt.

Um dieses Konzept zu vereinheitlichen, werden im Modul `pg542-util` fünf Schnittstellen zur Verfügung gestellt. Die Abkapslung des Lernprozesses durch diese Schnittstelle bietet auch die Möglichkeit Lernknoten besser untereinander zu vergleichen. Bei der Evaluation von Speicherverbrauch (siehe Kapitel 5) ist dies ein wichtiger Punkt.

Learner<T, M extends Model> Die wichtigste Schnittstelle stellt den eigentlichen Algorithmus dar, der zwei Aufgaben hat:

1. Lernen auf einem Element vom generischen Typ `T` (Eingabe)
2. Gebe auf den gelernten Elementen ein Modell `M` aus (Ausgabe)

```
1 public interface Learner<T, M extends Model> extends Serializable {
2   public void learn(T item);
3   public M getModel();
4 }
```

Abbildung 30: Generische Lernerschnittstelle

Der generische Typ `T` kann eine beliebige Klasse sein. Das Modell, das zurückgegeben wird, muss aus der Modellhierarchie stammen. Hier gibt es insgesamt vier Schnittstellen.

Model Dies ist die oberste Modellschnittstelle, die jedoch noch keine Methoden beinhaltet und lediglich als Markierungsschnittstelle dient. Jede weitere Modellschnittstelle erweitert dieses Interface.

PredictionModel<T, R> Bei dieser Schnittstelle stellt die schon erwähnten vorhersagenden Modelle dar. Das bedeutet, dass der Lerner aufgrund seiner Daten, die er erhalten hat, ein Modell erstellt, dass für ein weiteres Element vom Typ `T` eine Vorhersage vom Typ `R` tätigt (siehe Abbildung 31). Bei einer Klassifikation ist das beispielsweise die vorhergesagte Klasse.

```
1 public interface PredictionModel<T, R> extends Model {
2   R predict(T item);
3 }
```

Abbildung 31: Vorhersagendes Modell

DescriptionModel<R> Ein beschreibendes Modell dagegen besitzt nur ein generischen Typ, der den Rückgabewert beschreibt (siehe Abbildung 32). Dies könnte beispielsweise ein Histogramm sein, das die Häufigkeitsverteilung darstellt. Für viele beschreibende Modelle brauchen wir keine Parameter.

```
1 public interface {DescriptionModel<R> extends Model {
2   R describe();
3 }
```

Abbildung 32: Beschreibendes Modell

SelectiveDescriptionModel<T, R> Bei einem beschreibenden Modell gibt es aber auch Situationen, wo man zur Laufzeit die Ausgabe parametrisieren kann (siehe Abbildung 33). Beispielsweise könnte bei jeder Abfrage der häufigsten Elemente eine relative Mindesthäufigkeit angegeben werden.

```
1 public interface SelectiveDescriptionModel<T, R> extends Model {
2   R describe( T parameter );
3 }
```

Abbildung 33: Parametrisiertes, beschreibendes Modell

Zusammenfassend ist zu sagen, dass eine Implementierung eines Lernknoten durch abgekapselte *Learner*- und *Model*-Schnittstellen ein Vorteil ist, da es klar die algorithmische von der organisatorischen Seite eines Knoten trennt und damit die Wiederverwendbarkeit erhöht.

4.3.4 Parameter

Die Möglichkeit einen Knoten zu parametrisieren ist oft notwendig. Viele Knoten und Algorithmen benötigen eine Vielzahl von Parametern, die bei der Erstellung eines

Knotennetzwerkes angegeben werden müssen. Nun muss ein Knoten die Möglichkeit haben, zu sagen, welche Knoten er benötigt, damit das Framework überprüfen kann, ob gültige Parameter für diesen Knoten vorliegen, um notfalls den Benutzer darüber zu informieren.

Aus diesem Anlass kann ein Knoten *ParameterType*s definieren. Ein *ParameterType* definiert einen bestimmten Typ von Parameter, den dieser Knoten benötigt. Dieser Typ sagt aus, welche Klasse ein Parameter haben soll, oder ob er nur vordefinierte Werte zur Auswahl bietet, oder ob er optional ist und ein Default-Wert vorliegt, usw.

ParameterType Für die meisten primitiven Datentypen in Java gibt es eine *ParameterType*-Klasse mit unterschiedlichen Einstellungen. Unabhängig von diesen Klassen haben alle Parametertypen auch eine Menge von identischen Einstellungsmöglichkeiten. Für jeden der folgenden Werte gibt es in der Klasse *ParameterType* ein *set*-Methode.

Einstellung	**Beschreibung**
parameterKey	Der Schlüssel, als String, unter dem der Knoten diesen Parameter erwartet.
parameterClass	Der Klassentyp des Parameters. Wird der Parameter aus einer XML gewonnen, muss er sich zu diesem Datentyp umwandeln lassen. Folgende Klassen werden unterstützt: • Boolean • Integer, Long • Double, Float • String
description	Eine kurze Beschreibung des Parameters. Eine Beschreibung sollte üblicherweise gesetzt werden, weil ein Benutzer alleine durch der Parameterschlüssel nicht erkennen kann, für was dieser Parameter benötigt wird.
optional	Gibt an, ob der Parameter optional anzugeben ist. Falls er optional ist, sollte ein Default-Wert gesetzt werden.
defaultValue	Der Wert für diesen Parameter, der standardmäßig eingestellt ist.
predefinedValues	Wenn für diesen Parameter nur eine begrenzte Menge an Werten als Eingabe erlaubt sein soll, dann kann diese hier festgelegt werden.

ComparableParameterType Für vergleichbare Datentypen, also, Integer, Long, Double und Float, können zusätzlich noch Grenzen festgelegt werden zwischen denen sich der Parameter bewegen darf.

Einstellung	Beschreibung
lowerBound	Die untere Grenze (inklusive) des zulässigen Bereiches
upperBound	Die obere Grenze (inklusive) des zulässigen Bereiches

StringParameterType Handelt es sich bei den *ParameterType* um eine Zeichenkette, gibt es noch die Möglichkeit diesen String auf einen regulären Ausdruck zu testen (*setRegex()*). Für spezielle Parameter gibt es aber auch schon Klassen, die von *StringParameterType* erben und automatisch einen regulären Ausdruck setzen, wie z.B. *EmailParameterType*.

ListParameter Für jeden vorhandenen *ParameterType* gibt es auch eine Listenvariante mit den gleichen Einstellungsmöglichkeiten, nur dass nun eine Liste von Parametern übergeben werden kann. Die Werte werden dabei durch ein Trennzeichen (';') unterteilt.

4.4 Technische Details

Ideen, Anwendung und Implementierung von Knoten wurden in den letzten Kapiteln besprochen. Dies ist auch wichtig, um einen Einblick in das Framework zu erlangen. Möchte ein Entwickler jedoch am eigentlichen Kern des Frameworks weiter arbeiten, sind mehr Informationen hilfreich. In diesem Kapitel werden nun die technischen Details zur Umsetzung des Frameworks erläutert, um so einen guten Überblick zu erlangen.

Der Kern des Frameworks befindet sich vollständig im Projekt `pg542-core`. Alle konkreten Implementierungen, z.B. von Knoten, Algorithmen, aber auch von elementaren Bestandteilen wie der Evaluation, befinden sich im Projekt `pg542-util`. Das bedeutet, dass `pg542-core` nur die Grundfunktionalität des Frameworks darstellt. Diese Grundfunktionalität wollen wir nun näher untersuchen, jedoch ohne zu tief in den Programmcode zu gehen.

Wir beginnen womit wir im letzten Kapitel aufgehört haben und erklären, wie Knoten intern funktionieren.

4.4.1 Node

Bei der Implementierung unseres Knotens haben wir von der Klasse *AbstractNode* geerbt (siehe Kapitel 4.3). Diese Klasse ist aber nur eine Hilfsklasse, die dem Entwickler unter die Arme greift. Die kleinste gemeinsame Basis, die einen Knoten ausmacht, ist das *Node*-Interface. Jede Klasse, die dieses Interface nach Vereinbarung implementiert, kann als Knoten im Knotennetzwerk/-container eingesetzt werden. Es besteht aus ein paar wenigen Methoden, die zur Initialisierung, Parametrisierung und Ausführung eines Knotens notwendig sind.

NodeContext Ein wichtiger Unterschied zwischen der Schnittstelle *Node* und der abstrakten Klasse *AbstractNode* ist die Initialisierungsmethode. Ein Knotenentwickler implementiert diese parameterlos, wenn er von *AbstractNode* erbt. In der eigentlichen Schnittstelle wird aber ein *NodeContext*-Objekt übergeben. Jeder Knoten erhält eine eigene Instanz dieser Klasse, die dem Knoten ermöglicht mit seiner Umgebung zu kommunizieren. Dies bedeutet Parameter abrufen, Ausgaben weiterleiten, Überprüfung der Ein- und Ausgaben, Zugriff auf die internen Warteschlangen, usw.

Alle diese Methoden leiten den Aufruf an die zuständige Instanz weiter, also z.B. zum *NodeContainer*, *EventDispatcher* und *NodeProxy*. Der *NodeContext* verbirgt lediglich diese Abhängigkeiten vom Knoten und bietet eine einheitliche Schnittstelle. *AbstractNode* wiederum verbirgt dieses *NodeContext*-Objekt vor dem Knotenentwickler, indem es dieses in der *init*-Methode speichert und Methodenaufrufe an den *NodeContext* weiterleitet.

NodeProxy Entscheidend für einen richtigen Fluss von *Event*s ist, dass alle Knoten in einem eigenem Thread laufen. Wäre dies nicht so, dann könnte ein lokales Netzwerk immer nur ein *Event* gleichzeitig verarbeiten. Dies kann man mit dem Pipelining auf der Hardwareebene vergleichen. Ohne Pipelining (also ohne Bearbeitung von mehreren Befehlen auf unterschiedlichen Stufen gleichzeitig) ist der Durchsatz sehr viel geringer.

Umgesetzt wird dies durch eine Klasse namens *NodeProxy*. Für jeden Knoten, der in einem Container hinzugefügt wird, erstellt der Container ein Objekt dieser Klasse. *NodeProxy* bekommt die eigentliche Knoteninstanz und arbeitet als Proxy für diesen Knoten. Alle Methodenaufrufe des *Node*-Interface werden an den eigentlichen Knoten weitergeleitet. Aber anstatt beim Aufruf der *execute*-Methode diese Aufruf direkt weiterzuleiten, wird das *Event* in eine Warteschlange gespeichert. Beim Hinzufügen des Proxys zum Container wurde dieser als Thread gestartet, der in einer Endlosschleife diese Warteschlange abarbeitet. So benötigt das Weiterleiten eines *Event*s nur konstant viel Zeit, unabhängig davon, wie lange der Knoten für die Verarbeitung braucht. Der Container braucht den Knoten aber nach der Initialisierung nicht gesondert behandeln, da dieser ja die Knotenschnittstelle implementiert.

Trotzdem ist es manchmal sinnvoll auf *NodeProxy* spezifische Methoden zuzugreifen, um Zugriff auf diese Warteschlange zu erlangen. Beispielsweise kann man Informationen über die derzeitige Anzahl von *Event*s in der Warteschlange erlangen, diese bei Bedarf leeren oder sperren. Dies kann beim Herunterfahren eines Knoten hilfreich sein.

4.4.2 NodeContainer

Ein Knotencontainer ist eine lokale Ansammlung von Knoten (nicht unbedingt alle Knoten im Netzwerk), die von diesem Container verwaltet werden. Insofern implementiert ein Container eine *Collection* von Knoten und lässt sich auch fast vollständig so verwalten. Knoten werden also überwiegend hinzugefügt oder entfernt, oder es wird über die Knoten iteriert. Beim Hinzufügen von Knoten kümmert sich der Container darum, dass ein *NodeProxy* und ein *NodeContext* erstellt und in einem Thread gestartet werden und beim Entfernen wird darauf geachtet, dass ein Knoten sich ordnungsgemäß beenden kann.

Dies ist im Grunde genommen auch schon alles, was ein Container tut. Selbst das Weiterleiten von *Event*s wird hierbei vom *EventDispatcher* übernommen, der aber im Grunde genommen auch nur eine Sammlung von Knotenverbindung ist, die abgefragt werden, sobald ein Knoten die Methode *dispatch* aufruft. Der Knotencontainer ist trotzdem ein wichtiges Element im Framework, weil er als Bindeglied zwischen allen funktionalen Programmteilen fungiert.

NodeContainerHook Es gibt Situationen in denen die Funktionalität von einem Knotencontainer nicht ausreicht oder einige Beschränkungen zu sehr stören. In solchen Fällen ist es hilfreich den Container zu erweitern. Beispielsweise hat ein Knoten keine Möglichkeit auf Container-interne Informationen zuzugreifen. Dies ist auch vernünftig, denn meistens bringt es mehr Probleme als Vorteile mit sich, wenn Knoten z.B. direkt auf andere Knoten zugreifen könnten. Bei der Evaluation ist dies aber sinnvoll, wenn z.B. der Evaluationsknoten automatisiert den benutzen Speicher anderer Knoten messen möchte.

Diese Funktionalität können wir zur Verfügung stellen ohne den eigentlichen Code des Knotencontainers zu verändern. Dies passiert mit sogenannten Hooks. In einem Knotencontainer gibt es verschiedenste Methoden in denen bestimmte Phasen realisiert werden. Die Initialisierungsphase ist besonders bedeutsam, da wir hier die Interaktion zwischen Container und Knoten beeinflussen können, indem wir kurz vor oder kurz nach der Initialisierung zusätzlichen Code hängen. Dieser Code wird Hook genannt.

Für die Implementierung muss von der Klasse *NodeContainerHook* geerbt werden, die zwei leere Implementierungen für Methoden anbietet, die zum einen vor der Initialisierung und einmal danach aufgerufen werden. Als Parameter werden jeweils die

zu initialisierenden *NodeProxy*-Objekte (bzw. schon initialisierten) und der Knotencontainer übergeben, so dass hier die gewünschten Änderungen vollzogen werden können. Beispielsweise kann hier ein Hook den Evaluationsknoten suchen und ihm alle Knoten, die Lerner besitzen zur Evaluation injizieren.

Aktiviert wird der Hook in der Knotennetzwerk-Konfiguration. Im *NodeNetwork*-Interface muss die Methode *getHookClasses* eine Liste von implementierten Hook-Klassennamen zurückgeben, die in dieser Reihenfolge dann ausgeführt werden. In einer XML-Datei existiert dafür ein Tag namens `hook` in dem das Attribut `class` den vollständigen Klassenpfad zur der Hook-Implementierung angibt.

4.4.3 NamingService

Die Hauptaufgabe eines Namensdienstes besteht darin Knotenreferenzen anhand ihrer Id aufzulösen und somit eine Zugriffsmöglichkeit auf diesen Knoten zu ermöglichen. Dies passiert im einfachsten Fall lokal oder aber auch verteilt über ein größeres Netzwerk.

Die nächste wichtige Aufgabe ist die Verwaltung der Themenkanäle. Auch hier ist der Namensdienst dafür verantwortlich, dass sich Knoten zu Themen an- und wieder abmelden können oder die derzeit angemeldeten Knoten abgefragt werden, um etwas an diesen Themenkanal zu veröffentlichen.

All diese Funktionalität muss über die *NamingSerivce* Schnittstelle zur Verfügung gestellt werden. Wie diese Implementierung aussieht und mit welcher Technologie - insbesondere bei verteilten Netzwerken - diese umgesetzt wird, ist unerheblich. Wichtig ist nur, dass sich die Implementierung an die Vereinbarungen der Schnittstelle hält.

Lokaler Namensdienst Eine Implementierung der Schnittstelle, die schon im Framework enthalten ist, ist die eines lokalen Namensdienstes. Dieser Namensdienst ist dafür geeignet, wenn ein Knotennetzwerk nur lokal in einem Container läuft. Er besteht prinzipiell nur aus einigen wenigen *Map*s, die die Knoten und Themenkanäle verwalten und bietet so einen schnellen und einfachen Namensdienst. Diese Klasse kann auch in verteilten Implementierungen wiederbenutzt werden, um lokale Informationen zu speichern.

Verteilter Namensdienst Soll ein Knotennetzwerk verteilt auf mehreren physikalischen Maschinen laufen, muss die Implementierung des Namensdienstes dies unterstützen. Die Schnittstellenvereinbarung sieht Folgendes vor:

- Es gibt für jeden Namensdienst einen oder keinen übergeordneten Namensdienst.

- Bei einer Namensauflösungsanfrage wird zunächst lokal gesucht, ansonsten wird die Anfrage zum übergeordneten Namensdienst propagiert.
- Wird ein Knoten nicht gefunden und gibt es keinen übergeordneten Namensdienst mehr, dann wird eine *NamingException* geworfen.
- Alle Anfragen zu Themenkanälen und zur Registrierung von Knoten werden solange an den übergeordneten Namensdienst weitergeleitet, bis dieser nicht mehr existiert. Dieser gefundene Namensdienst besitzt dann alle notwendigen Informationen.

Durch diese Vereinbarung entsteht ein Baum aus Namensdiensten (siehe Abbildung 34). In diesem Baum besitzt der oberste Knoten alle Knoteninformationen im gesamten Netzwerk. Bei Auflösungsanfragen wird in diesem Netzwerkbaum solange gesucht, bis die benötigte Information gefunden wurde, bei An- und Abmeldungen bzw. Veröffentlichungen auf Themenkanälen wird immer bis zum obersten Knoten weitergeleitet.

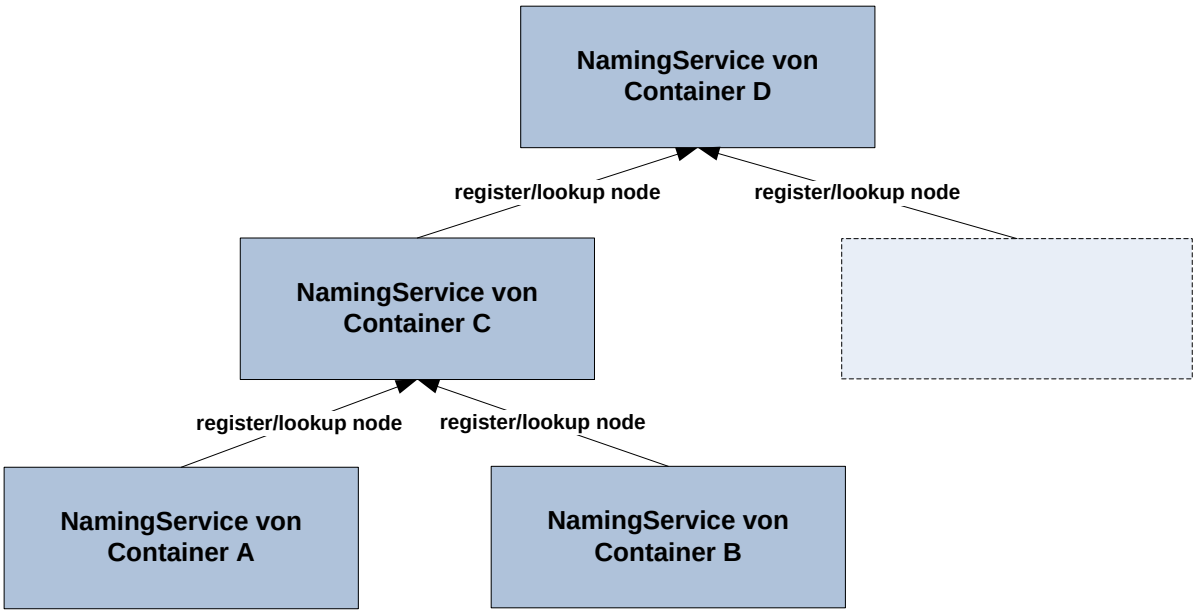


Abbildung 34: Baum aus Namensdiensten

Die Umsetzung dieser Namensdienste steht vollkommen frei. Da der Namensdienst jedoch für die gesamte Kommunikation zuständig ist - also auch für den Transport der *Event*s - ist die Verwendung einer Technologie, die entfernte Methodenaufruf unterstützt, äußerst sinnvoll. Zu nennen sei hier das sprachenunabhängige `CORBA` oder `RMI`, das mit Java ausgeliefert wird.

RMI Namensdienst Neben dem lokalen Namensdienst beinhaltet das Framework eine Implementierung für den verteilten Einsatz mit Hilfe von `RMI` (Remote Method

Invocation). RMI bietet die Möglichkeit Methoden auf bestimmten registrierten Objekte entfernt aufzurufen. Das folgende Kapitel setzt grundlegende Kenntnisse über RMI voraus.

Zunächst gehen wir auf die Kommunikation zwischen zwei Knoten über RMI ein, da dies dem Verständnis zur eigentlichen Namensdienst-Implementierung beisteuert.

Bei der Kommunikation ist der *EventDispatcher* dafür zuständig *Event*s weiterzuleiten. Für ihn ist nur wichtig, dass der Knoten, an den er das *Event* schicken soll, dass *Node*-Interface implementiert und nicht, ob dieser Aufruf nun lokal oder entfernt stattfindet. Wir benötigen also auch hier wieder ein Knotenproxy, der die Aufrufe des *EventDispatcher* annimmt und diesen nun nicht an einen lokalen Knotenthread weiterleitet (siehe Kapitel 4.4.1), sondern über RMI an einen entfernten Knoten. Dieser Proxy heißt *SenderNodeProxy*. Bei einem Aufruf der *execute*-Methode baut dieser Proxy eine Verbindung zu einem entfernten - in seiner RMI-Registry registrierten - Objekt der Klasse *ReceiverNodeRemoteObject* auf. Jedes dieser Objekte repräsentiert einen Knoten, deren Objektreferenz er besitzt und leitet die entfernten Aufrufe weiter (siehe Abbildung 35).

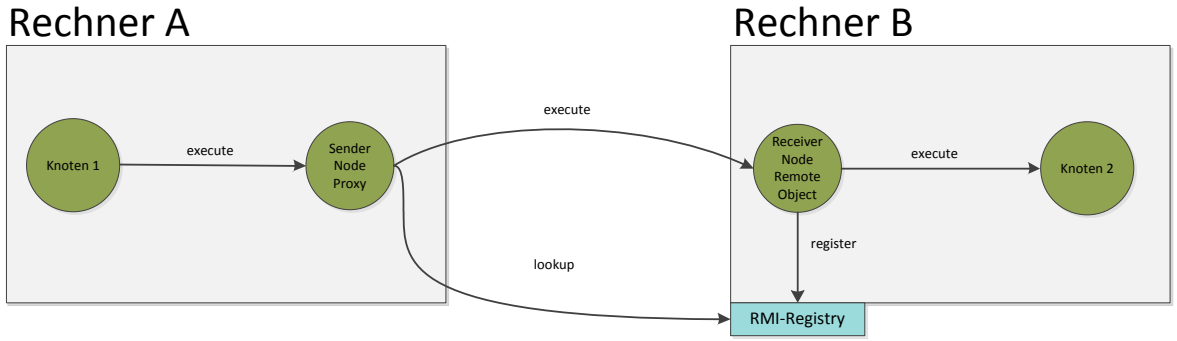


Abbildung 35: Kommunikation zwischen zwei Knoten

Das *SenderNodeProxy*-Objekt selbst ist serialisierbar, da er beim Versenden nur die IP-Adresse der RMI-Registry benötigt und die Knoten-Id unter der der Knoten dort registriert ist. Mit diesen Informationen kann eine Verbindung aufgebaut werden. Zur Auflösung einer Knotenreferenz eines entfernten Knoten, muss der Namensdienst also nur das *SenderNodeProxy*-Objekt zurückgeben, der die Informationen zu einem registrierten (bzw. zu einem Thema angemeldeten) *ReceiverNodeRemoteObject* beinhaltet.

Für die Verwaltung dieser *SenderNodeProxy*s registriert jeder Namensdienst ein Objekt der Klasse *NamingServiceRemoteObject* in der RMI-Registry. Dieses Objekt ist die Schnittstelle zwischen lokalen und entfernten Aufrufen. Es empfängt Knotenregistrierungen oder Anmeldungen zu Themenkanälen, speichert *SenderNodeProxy*s lokal, um sie bei einer Anfrage zurückzugeben, und leitet diese Anfragen an das übergeordnete *NamingServiceRemoteObject* weiter.

Die Aufrufe an *NamingServiceRemoteObject* erfolgen also einerseits durch den lokalen Container oder durch einen anderen Namensdienst. Das *NamingServiceRemoteObject* unterscheidet nicht zwischen den verschiedenen Aufrufern. Die Aufrufe vom lokalen Container müssen aber noch Vorkehrungen zur entfernten Registrierung treffen. Das bedeutet, wir müssen das *ReceiverNodeRemoteObject* erstellen und registrieren, ein passenden *SenderNodeProxy* erstellen und dann an *NamingServiceRemoteObject* weiterleiten, so dass es hierfür keinen Unterschied macht, ob der *SenderNodeProxy*-Objekt nun von einem entfernten Namensdienst stammt oder vom lokalen Container. Für genau diese Aufgabe ist ein *LocalToRemoteAdapter*-Objekt zuständig.

Damit haben wir fast das komplette Grundgerüst für unsere `RMI`-Namensdienstimplementierung gelegt. Was wir aber noch vermeiden wollen, ist der Umstand, dass nach dem bisherigen Aufbau lokale Verbindungen auch über `RMI` ablaufen würden. Diesen Overhead wollen wir vermeiden. Hierfür benutzen wir eine *LocalNamingServiceFacade*, die wir dann letztendlich dem Knotencontainer übergeben. Dieses Fassaden-Objekt überprüft für jede Anfrage, ob es sie auch lokal über ein *LocalNamingService* auflösen kann und reicht den Aufruf ansonsten an den *LocalToRemoteAdapter* weiter.

Der vollständige Objektgraph ist in Abbildung 36 zu sehen.

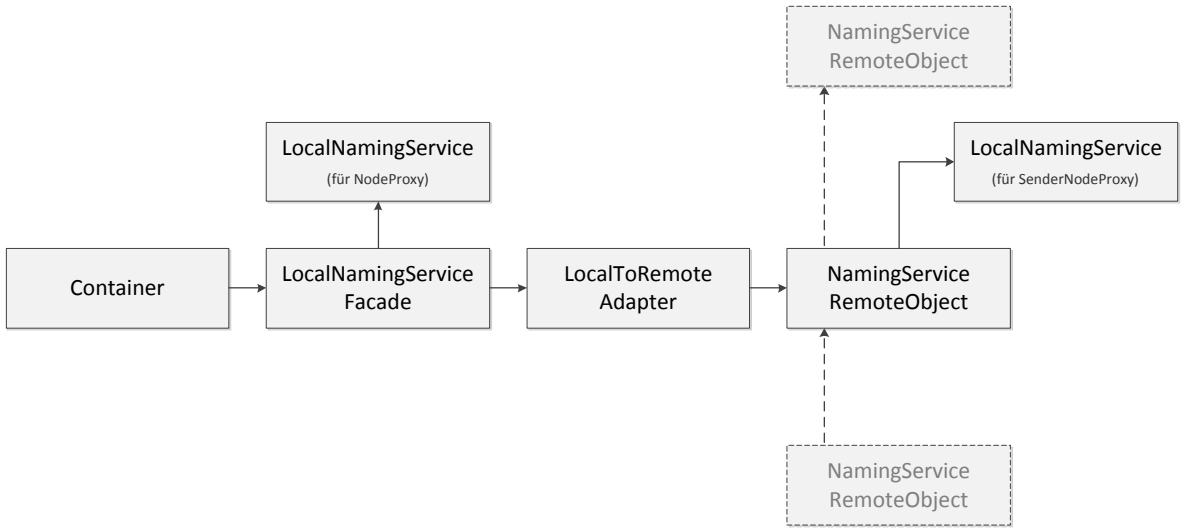


Abbildung 36: Namensdienstimplementierung über RMI

4.4.4 Bootstrap

In diesem letzten Kapitel behandeln wir, was beim Booten eines Containers intern passiert. Diesen Vorgang übernimmt *ContainerBootstrap* indem es eine *NodeNetwork*-Instanz erhält und beim Aufruf der *boot*-Methode verschiedene Schritte ausführt:

container-instantiation In diesem Schritt wird aus dem *NodeNetwork*-Objekt der vollständige Klassenname für die *NamingServiceFactory*, inklusive IP-Adresse und Id extrahiert, um dann damit den Knotencontainer zu instantiieren.

node-instantiation Alle Knoten werden anhand ihres vollständigen Klassennamens instanziiert.

node-injection Alle instantiierten Knoten werden in den Container injiziert, d.h. dem Container hinzugefügt, der dann die jeweiligen *NodeContext*- und *NodeProxy*-Objekte erstellt und verwaltet.

parameter-processing Aus allen Knoten werden die *ParameterType*-Objekte gesammelt und gegenüber den vorliegenden Parametern aus dem *NodeNetwork*-Objekt validiert. Ist diese Evaluation erfolgreich, dann werden die Parameter in den jeweiligen Knotenkontext injiziert.

node-initialization Die *init*-Methode von jedem Knoten wird aufgerufen.

node-linking Alle Verbindungen werden ausgelesen, überprüft und gesetzt

container-cascading Wenn ein übergeordneter Namensdienst gesetzt wurde, dann wird dieser dem lokalen Namensdienst mitgeteilt.

configure Global Containerkonfiguration werden ausgelesen und gesetzt, also beispielweise die Messung der lokalen und entfernten Kommunikation.

Nach diesen Schritten wird ein vollständig einsatzbereiter Knotencontainer zurückgegeben, der schon von anderen Containern über deren Namensdienste aufgerufen werden kann. Um die `kickoff`-Knoten von diesem Container jetzt noch zu starten, muss das *ContainerBootstrap*-Objekt die *run*-Methode mit diesem Container als Parameter ausführen. Alle gesetzten `kickoff`-Knoten werden damit gestartet.

5 Evaluation

5.1 Einleitung

Die im Rahmen der Projektgruppe betrachteten und implementierten generischen Verfahren sind im Wesentlichen in drei Problembereiche einzuordnen: Klassifikation, Regression / Zählen und Clustering. Um für diese Verfahren sowohl die Güte der resultierenden Modelle, als auch Daten wie Durchsatz, Speicherverbrauch und Zeit der implementierten Lerner zu messen und somit Verfahren innerhalb eines Themengebietes miteinander vergleichbar zu machen, wurden innerhalb des Frameworks Evaluationsmechanismen entwickelt, die dies ermöglichen sollen.
Die Notwendigkeit von Evaluationsverfahren ist dabei recht einfach motivierbar. Ein Unternehmen käme wohl niemals auf die Idee sein bisheriges Verfahren durch ein neues zu ersetzen, wenn nicht aussagekräftige Kennzahlen für eine Verbesserung durch eben jenes neue Verfahren sprechen würden. Ganz ähnlich sieht es im wissenschaftlichen Bereich aus, wo neuartige Verfahren in Relation zu bisherigen gebracht werden um dadurch Verbesserungen auszudrücken.
Ein besonderer Schwerpunkt wurde bei der Umsetzung auf die Evaluation der Modellgüte gelegt. So ist es möglich, nach Belieben Gütemaße zu implementieren und für die Evaluation der Modellgüte zu verwenden.
Es folgt eine Betrachtung der drei genannten Problembereiche, sowie eine Erläuterung der Abbildung der Evaluation von Modellgüte, Speicherverbrauch, Zeit und Durchsatz auf das Framework.

5.2 Evaluation von Klassifikationsproblemen

5.2.1 Motivation

Bei Klassifikationsproblemen hat man die Situation, dass ein Datensatz vorliegt, dessen Verwendung in Form von *Trainingsdaten* und *Testdaten* vonstattengeht. Die Trainingsdaten dienen der Erzeugung eines Modells, auf dessen Basis zukünftige Klassifikationsvorhersagen gemacht werden können, während die Testdaten zur Überprüfung der Güte des erzeugten Modells herangezogen werden.
Im Folgenden werden einige Genauigkeitsmaße zur Evaluation der Modellgüte von Klassifikationsverfahren dargestellt. Außerdem werden vier verschiedene Strategien zur Verwendung eines Datensatzes als Trainings- und Testdaten behandelt: *Holdout*, *Bootstrap*, *Crossvalidation* und *Interleaved-Test-Then-Train*. Da es sich bei dem vorliegenden Framework um ein Streammining-Framework handelt, werden auch die speziellen Probleme angesprochen, die es in diesem Fall zu beachten gilt.
Alle im Folgenden aufgeführten Beispiele sind in dieser oder ähnlicher Form in [22, 6, 41] beschrieben.

5.2.2 Genauigkeitsmaße

Als Hauptmaß der Evaluation bei Klassifikationsproblemen wird die Genauigkeit betrachtet. Dabei meint man die Anzahl der richtig klassifizierten Beispiele eines Testdatensatzes im Verhältnis zu allen gesehenen Beispielen. Ein anderes häufig verwendetes Maß in diesem Zusammenhang ist die Fehlerrate. Dabei handelt es sich um nichts anderes als $1 - Genauigkeit$.
Die sogenannte *Konfusionsmatrix* ist eine geeignete Datenstruktur, mit deren Hilfe analysiert werden kann, wie gut oder schlecht ein bestimmter Klassifizierer die Zugehörigkeit eines Beispiels zu einer Klasse bestimmt. In Abbildung 37 ist eine allgemeine Konfusionsmatrix für zwei unterschiedliche Klassen gezeigt. Wie man sieht gibt es vier verschiedene Kategorien, in welche die Klassifikation der Beispiele einsortiert werden kann. Am Beispiel aus Tabelle 1, in welchem gezeigt ist, wie viele Angriffe auch tatsächlich als solche erkannt wurden bzw. wie viele Nichtangriffe fälschlicherweise als Angriffe klassifiziert wurden etc. lässt sich erkennen, dass ein Klassifizierer mit guter Genauigkeit die meisten Beispiele in die Hauptdiagonale einsortiert, während die anderen Felder Werte nahe der Null enthalten sollten. Im Allgemeinen handelt es sich bei der Konfusionsmatrix um eine $n \times n$ Matrix, wobei für jede vorkommende Klasse eine Zeile und eine Spalte reserviert werden.

		Actual class	
		C1	C2
Predicted class	C1	true positives	false negatives
	C2	false positives	true negatives

Abbildung 37: Konfusionsmatrix

Klasse	Ang.erkannt = ja	Ang.erkannt = nein	Summe	Erk.rate (%)
erkannt = ja	6.954	46	7.000	99.34
erkannt = nein	412	2.588	3.000	86.27

Tabelle 1: Beispiel:Konfusionsmatrix

Manchmal interessiert man sich nur für eine bestimmte Klasse von Daten. In diesem Fall ist die Genauigkeit kein gutes Gütemaß. So gibt es viele Anwendungsfälle, in denen man sich eher für die Minorität einer Klasse interessiert. Intrusion Detection stellt dabei ein ideales Beispiel dar. Angenommen, man hätte ein Modell gelernt, das zwischen Angriff und normalem Zugriff unterscheiden soll. Dabei werden Genauigkeitswerte von 90% erreicht, was für sich genommen schon recht gut ist. Wenn jedoch

in den Trainingsdaten, aus denen dieses Modell gelernt wurde, nur 3% der Zugriffe Angriffe waren, würde das Modell nur die normalen Zugriffe zu ca. 90% klassifizieren. In diesem Fall interessiert man sich viel mehr dafür, wie gut ein Klassifizierer erkennen kann, dass es sich nicht um einen Angriff handelt. Aus diesem Grund gibt es die sog. *sensitivity* und die *specificity*. Mit der sensitivity - auch *Recall* genannt - (Gleichung 3) kann die Rate der "true positives" ermittelt werden, während die specificity (Gleichung 4) die "true negative" Rate wiedergibt.

$$sensitivity = \frac{TP}{TP + FN} \tag{3}$$

$$specificity = \frac{TN}{FP + TN} \tag{4}$$

$$precision = \frac{TP}{TP + FP} \tag{5}$$

Mit diesen Werten lassen sich verschiedene Aussagen über die Güte der Resultate eines Klassifizierers treffen. Bildet das Klassifikationsproblem Daten auf mehr als zwei Klassen ab, wird wie bereits erwähnt für jede Klasse eine Zeile und eine Spalte in der Konfusionsmatrix erzeugt. Für die Berechnung der soeben eingeführten Werte in Bezug auf eine Klasse werden aus der Konfusionsmatrix die Werte so extrahiert, dass sie wieder als 2 x 2 Konfusionsmatrix dargestellt werden können. Dies ist in Abbildung 38 beispielhaft für die Klasse C1 in einer 3 x 3 Konfusionsmatrix gezeigt.

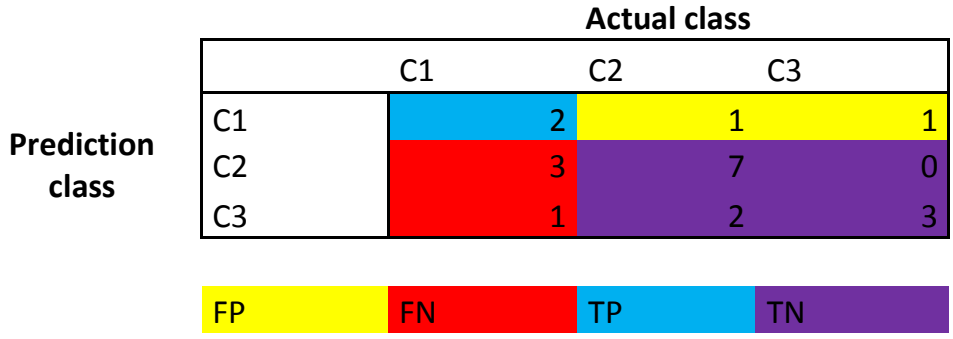


Abbildung 38: Table Of Confusion

Wie oben bereits erwähnt, sollte man zu einer gegebenen Klassifikationsaufgabe die „beste" Methode verwenden. Es stellt sich also die Frage, nach welchem Kriterium man entscheiden kann, dass ein Klassifizierer besser ist als ein anderer. Die beiden Maße Precision und Recall können diese Frage nicht beantworten, da es sich hierbei um zwei funktional unterschiedliche Maße handelt und somit eine qualifizierte

Aussage nicht möglich ist. Besser geeignet ist hier der sog. **F-score**. Dieser ist als das harmonische Mittel von der Precision und Recall definiert. Sowohl Gleichung 6 als auch 7 beschreiben diesen.

$$F - Score = \frac{2pr}{p + r} \tag{6}$$

$$F - Score = \frac{2}{\frac{1}{p} + \frac{1}{r}} \tag{7}$$

5.2.3 Methoden

Im Folgenden werden die eingangs erwähnten Strategien zur Gewinnung von Trainings- und Testdaten aus einem Datensatz vorgestellt. Hierzu ist es in jedem Fall erforderlich, dass die Elemente des Datensatzes bereits klassifiziert sind, da ansonsten die Einordnung in die Konfusionsmatrix nicht möglich ist und somit auch keine Evaluation der Modellgüte erfolgen kann.
Alle erläuterten Strategien haben unter anderem das gemeinsame Ziel, nur solche Daten als Testdaten zu verwenden, die nicht im Training verwendet wurden. Andernfalls würde die Gefahr bestehen, dass das erzeugte Modell ausschließlich für die Trainingsdaten gut, für alle weiteren Daten jedoch schlecht ist. Dies wird in der Fachliteratur als *overfitting* bezeichnet. Die umgekehrte Situation - erst Testen, dann Trainieren - ist jedoch nicht ausgeschlossen. Somit resultiert die Aufteilung in Trainingsdaten und Testdaten nicht notwendigerweise in disjunkten Mengen.

Holdout Gegeben sei eine Datenmenge D, welche in Trainings- und Testdaten in Form zweier disjunkter Teilmengen aufgeteilt wird. Es ist also $D = D_{train} \cup D_{test}$ und $D_{train} \cap D_{test} = \emptyset$. Diese Methode kommt häufig dann zum Einsatz, wenn der Datensatz groß ist.
Bei der Frage der gerechten Aufteilung in Trainings- und Testdatensatz gibt es verschiedene Ansätze. Die Größe des Datensatzes spielt dabei eine entscheidende Rolle. In der Literatur findet man häufig die $50/50$ Aufteilung, d.h. dass jeweils die Hälfte des Datensatzes als Trainingsdaten und die andere Hälfte als Testdaten verwendet wird. Eine andere, ebenfalls häufig verwendete Möglichkeit ist eine $\frac{2}{3}/\frac{1}{3}$ Verteilung zugunsten der Trainingsdaten.
Auch bei der Auswahl der Beispiele, die den oben erwähnten Teilmengen zugeordnet werden, gibt es diverse Möglichkeiten wie diese ausgewählt werden können.

1. Randomisierte Auswahl der Trainingsdaten aus D, wobei die übrigen als Testdaten verwendet werden.
2. Häufig kommen Beispiele zeitversetzt an. Dieser Umstand spiegelt auch eher den Charakter einer realen Applikation wieder. Hierbei ist es also möglich die zuerst gesehen Beispiele für das Training zu nutzen.

Falls der Datensatz zu klein sein sollte, kann es passieren, dass diese beiden Methoden nicht zu sinnvoll zu gebrauchen sind, da keine repräsentative Aussage getroffen werden kann. In solchen Fällen ist es möglich das bei Punkt 1. erwähnte randomisierte Verfahren n-Mal anzuwenden, wobei jedes Mal ein anderer Trainings- und Testdatensatz entsteht. Damit werden auch n unterschiedliche Genauigkeitswerte gebildet. Zu guter Letzt kann man durch die Bildung des Durchschnitts über alle Werte einen endgültigen Wert erhalten.
Im Framework selbst wurde das Holdout Verfahren wie in Abbildung 39 entworfen und umgesetzt. Dort sieht man eine Batch-Version dieses Ansatzes. Erwähnenswert hierbei ist noch das sogenannte *Stratifizieren*. Dieser Ausdruck beschreibt, wie die Beispiele mit ihren Attributen bei der oben erwähnten Aufteilung in Trainings- und Testdaten aufgeteilt werden. So ist häufig eine approximierte Gleichverteilung der Attribute wünschenswert. Dies wird durch die Stratifizierung erreicht. Ohne diese kann es passieren, dass die Trainings- wie auch die Testdaten gänzlich unterschiedlich Attribute aufweisen und die Ergebnisse womöglich verfälscht werden. Auf der anderen Seite kann bei homogenen Daten der Aufwand der für das Stratifizieren entsteht eingespart werden.

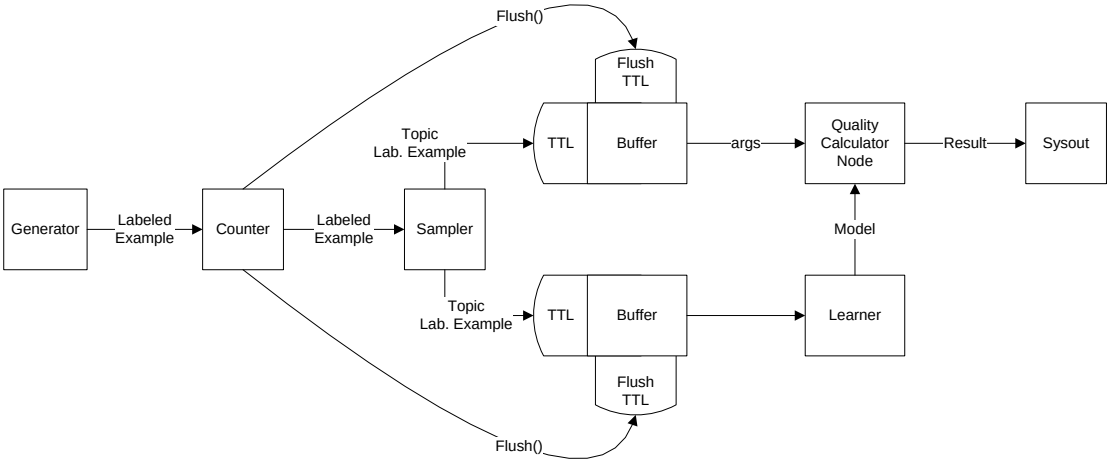


Abbildung 39: Holdout (Batch)

In Abbildung 40 sieht man den Entwurf des Holdout Verfahrens für Streams. Dabei lässt sich erkennen, dass das Grundprinzip gleich geblieben ist. Die wichtigste Änderung ist, dass in der Streamversion kontinuierlich neue Beispiele an den Lerner gereicht werden und so eine regelmäßige Güteevaluation stattfinden kann, während in der Batchversion der Vorgang des Evaluierens nur ein einziges Mal stattfindet.

Crossvalidation Diese Strategie der Datenaufteilung kommt häufig zum Einsatz, wenn der zur Verfügung stehende Datensatz klein ist. Bei diesem Verfahren werden

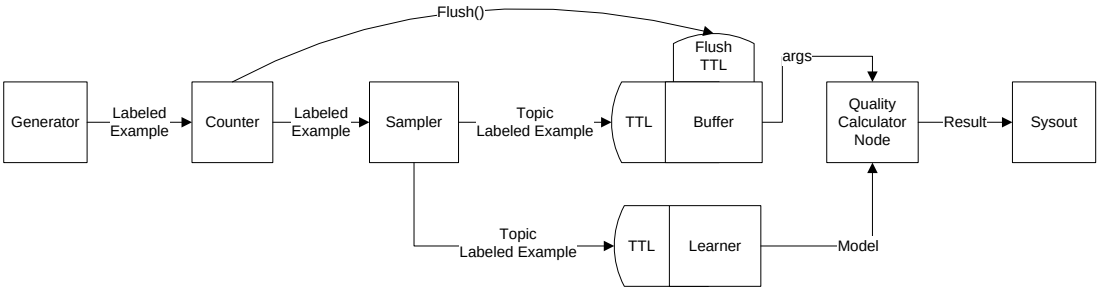


Abbildung 40: Holdout (Stream)

die Daten in k gleichgroße, disjunkte Datensätze partitioniert. Es finden dann k Iterationen der Evaluation statt, wobei jede der k Partitionen dabei einmal als Testmenge verwendet wird, während die Vereinigung der restlichen $k-1$ Partitionen die Trainingsdaten dieser Iteration darstellt. Auf diese Weise entstehen k Ergebnisse, welche anschließend zu einem Ergebnis gemittelt werden. Laut Literatur hat sich die 5-fach sowie die 10-fach Crossvalidation als quasi-Standard etabliert.
In Abbildung 41 ist das beschriebene noch einmal grafisch veranschaulicht.

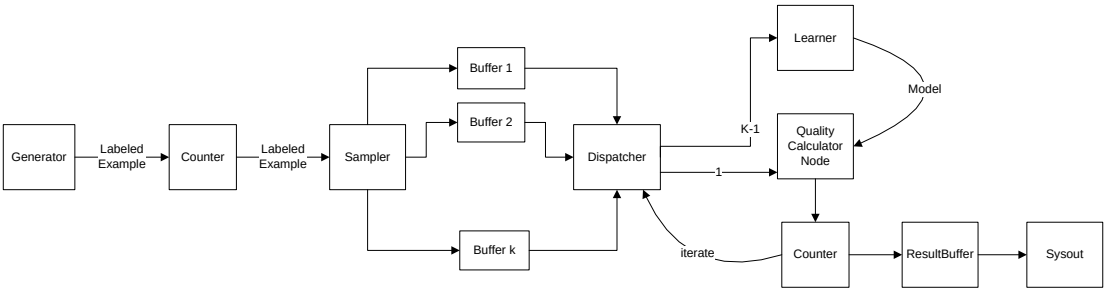


Abbildung 41: Crossvalidation (Batch)

Leave-one-out -Crossvalidation ist ein Spezialfall der soeben beschriebenen Crossvalidation. Hierbei wird die Anzahl k der Partitionen als die Größe des Datensatzes gewählt, was zur Folge hat, dass in einer Iteration der Evaluation nur ein einziges Beispiel aus dem Datensatz für das Testen verwendet wird und alle anderen der Erzeugung eines Modells dienen. Dieses Verfahren eignet sich nur, falls der Datensatz sehr klein ist, da der Berechnungsaufwand hierbei erheblich steigt.

Bootstrap stellt eine Methode dar, mit der ein Trainings- und ein Testdatensatz in der Form erzeugt werden, dass der Trainingsdatensatz genauso viele Elemente enthält wie der ursprüngliche Datensatz. Grob gesagt handelt es sich hierbei um das Prinzip des „Ziehens mit Zurücklegen“. Der Vorgang kann wie folgt beschrieben werden:

Aus einer Datenbasis mit n Elementen wird zufällig n Mal ein Element gewählt und in den Trainingsdatensatz kopiert. Diese wird anschließend wieder zurückgelegt. In dem so erzeugten neuen Trainingssatz sind einige Elemente mehrfach vorhanden. Im Umkehrschluss muss es in der ursprünglichen Menge noch Elemente geben, welche nicht im Trainingsdatensatz vertreten sind. Diese werden als Testdaten gewählt. In Abbildung 42 wird dieser Umstand dadurch erreicht, dass Elemente, die schon einmal zum Trainingsdatensatz hinzugefügt worden sind, markiert werden. Alle nicht markierten Elemente können am Ende für Testzwecke verwendet werden.
Diese Strategie wird häufig als 0.632 Bootstrap bezeichnet. Die Bezeichnung rührt daher, dass für jedes einzelne Element die Wahrscheinlichkeit **nicht** im Trainingsdatensatz zu landen bei $1 - \frac{1}{n}$ liegt. Das wiederum multipliziert mit der Anzahl der Auswahlvorgänge n ergibt: $(1 - \frac{1}{n})^n \approx e^{-1} = 0.368$. Für eine ausreichend große Datenmenge fallen somit 36,8% aller Instanzen in die Testmenge und 63,2% in die Trainingsmenge.

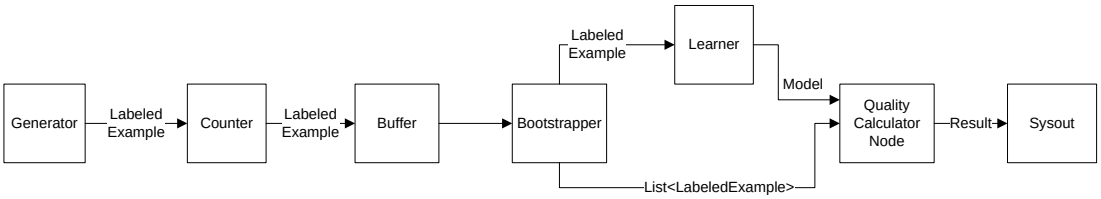


Abbildung 42: Bootstrap (Batch)

Interleaved Test-Then-Train ist eine Strategie für den Einsatz im Streammining-Kontext. Jeder eingehende Datenpunkt wird zunächst für das Testen verwendet und erst im Anschluss benutzt, um das Modell zu verfeinern, was den Vorteil bietet, sämtliche Daten für Training und Test verwenden zu können. Dieses Verfahren eignet sich am besten, wenn die Evaluation sehr genau und so häufig wie möglich aktualisiert werden soll. Der schematische Aufbau ist in Abbildung 43 zu sehen.
Obwohl auf den ersten Blick dieses Verfahren vermutlich am sinnvollsten erscheinen mag, so besteht hier die Gefahr, dass vor allem im Anfangsstadium, in dem das Modell noch nicht sehr ausgereift ist, viele Klassifikationsfehler entstehen können. In der Grafik ist auch ersichtlich, dass zu Beginn eine gewisse Vorlaufzeit benötigt wird, in der alle Beispiele nur für das Lernen des Modells verwendet werden und noch keine Tests stattfinden.

Anmerkung: All diese Verfahren können im Framework durch eine geeignete Konfiguration und Kombination bereits existierender Knoten für eigene Lerner nachgestellt werden. Für die gängige Verwendung stehen Klassen bereit, mit der der Konfigurationsaufwand auf ein Minimum reduziert werden kann. Es sei jedoch darauf hingewiesen, dass nicht jeder Spezialfall hierbei berücksichtigt werden kann.

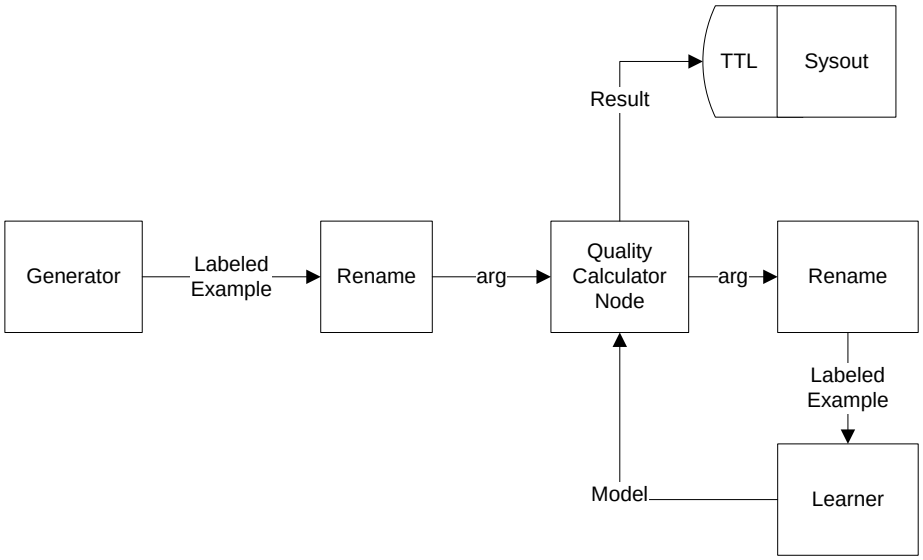


Abbildung 43: Interleaved Test-Then-Train

5.3 Evaluation von Regressionsproblemen

5.3.1 Motivation

Bei der Regression handelt es sich um eine statistische Methode zur Vorhersage von kontinuierlichen Werten. Häufig werden die Begriffe Regression und numerische Vorhersage parallel verwendet.
Die Regressionsanalyse kann dazu verwendet werden Beziehungen zu modellieren, die zwischen einer oder mehreren unabhängigen Variablen - auch *Predictoren* genannt - und einer abhängigen Variable bestehen. Im Allgemeinen sind die Werte der Predictoren bekannt.
Im Folgenden werden die wichtigsten Arten von Regressionsanalysen beschrieben.

5.3.2 Lineare Regression

Bei der linearen Regression geht man davon aus, dass eine Variable y, sowie eine einzige Predictor Variable x vorhanden sind. Es handelt sich dabei um die einfachste Form der Regression. Hierbei wird y als eine lineare Funktion von x abgebildet. Gleichung 8 beschreibt dabei die allgemeine Form der Funktion. Dabei können die Koeffizienten b und w auch als Gewichte angesehen werden. Diese können mit der Methode der kleinsten Quadrate gelöst werden. Die Menge D ist hierbei erneut der Trainingsdatensatz, welcher jeweils aus Tupeln der Form: $(x_1, y_1, ..., (x_{|D|}, y_{|D|})$ besteht. $\bar{x}$ ist hierbei der Mittelwert aller x-Werte und $\bar{y}$ der der jeweiligen y-Werte.

$$y = b + wx \tag{8}$$

$$w = \frac{\sum_{i=1}^{|D|}(x_i - \bar{x})(y_i - \bar{y})}{\sum_{i=1}^{|D|}(x_i - \bar{x})^2} \tag{9}$$

Durch ein Beispiel soll dies nochmal verdeutlicht werden. Abbildung 45 zeigt den zugehörigen Graphen für die Daten der Tabelle 44. Dabei handelt es sich um das Gehalt eines Mitarbeiters, das sich - abhängig von den Jahren die er in einem Unternehmen gearbeitet hat - verändert. Die Gerade die man dort sieht, erhält man, wenn man das Ergebnis aus Gleichung 9 in die Gleichung 8 einsetzt und diese nach b auflöst.

x Jahre	y Gehalt
3	30
8	57
9	64
13	72
3	36
6	43
11	59
21	90
1	20
16	83

Abbildung 44: Gehaltsdaten

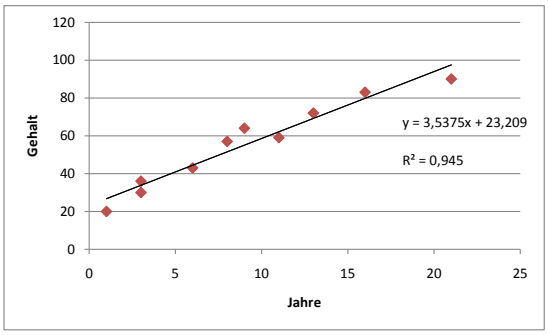


Abbildung 45: Grafik zur Gehaltsdatentabelle

Weiterhin gibt es die *multiple* lineare Regression, wobei es sich hierbei um eine Erweiterung der linearen Regression handelt. Dabei geht es darum mehr als einen Predictor einzubeziehen. Die Gleichung 8 kann dabei auf mehrere Variablen erweitert werden, wobei der Lösungsweg sich nicht vom obigen unterscheidet.

5.3.3 Fehlermaße für Vorhersagewerte

Bei der Frage nach der Akkuratheit einer Vorhersage erhält man als Rückgabewert kontinuierliche numerische Werte an Stelle von kategorisierten Labels zurück. Man bedient sich hierbei verschiedener Fehlermaße, die angeben, wie stark das vorhergesagte Ergebnis vom wahren Ergebnis abweicht. Neben den bereits in Tabelle 2 erwähnten Fehlermaßen finden sich noch diverse andere in der Fachliteratur. Dabei geht man von folgender Situation aus: Die Menge D bezeichnet den Testdatensatz der Form $(X_1, y_1), \ldots, (X_d, y_d)$. X_i sind dabei die n-dimensionalen Testtupel mit den zugehörigen Werten y_i und d ist die Anzahl der Tupel in D.

Fehlermaß	Formel
Absoluter Fehler	$\lvert y_i - y_i^{'} \rvert$
Quadratischer Fehler	$(y_i - y_i^{'})^2$
Mittlerer absoluter Fehler	$\frac{\sum_{i=1}^{d} \lvert y_i - y_i^{'} \rvert}{d}$
Mittlerer quadratischer Fehler	$\frac{\sum_{i=1}^{d} (y_i - y_i^{'})^2}{d}$
Relativer absoluter Fehler	$\frac{\sum_{i=1}^{d} \lvert y_i - y_i^{'} \rvert}{\sum_{i=1}^{d} \lvert y_i - \overline{y}_i \rvert}$
Relativer quadratischer Fehler	$\frac{\sum_{i=1}^{d} (y_i - y_i^{'})^2}{\sum_{i=1}^{d} (y_i - \overline{y}_i)^2}$
mit	$\overline{y} = \frac{\sum_{i=1}^{d} y_i}{d}$

Tabelle 2: Predictor Fehlermaße

Dabei bezeichnet y_i den tatsächlichen Wert und $\overline{y}_i$ den vorhergesagten Wert. Beim mittleren quadratischen Fehlerwert fallen Ausreißer mehr ins Gewicht als beim mittlere absoluten Fehler. Auch ist es möglich zusätzlich die Wurzel des quadratischen Fehlers zu betrachten, wodurch der berechnete Wert im gleichen Maß dargestellt werden kann wie die Vorhersage.

Anmerkung: In der Praxis hat die Wahl der Fehlermetrik häufig keinen starken Einfluss auf die Wahl des Vorhersagemodells.

5.4 Evaluation von Clusteringproblemen

5.4.1 Motivation

Nachdem es gelungen ist, mit geeigneten Clustering-Algorithmen die vorhandenen Daten Clustern zuzuordnen, ist man natürlich auch daran interessiert, wie gut bzw. schlecht diese Zuordnung ist. Ganz anders als bei Klassifikationsproblemen, wo es gelabelte Daten gibt und eine Gütemessung somit relativ leicht erscheint, gibt es diesen Vorteil beim Clustering nicht. Im Folgenden werden einige der gebräuchlichsten Methoden erläutert die in der Literatur zu finden sind.

5.4.2 Methoden

Manuelle Sichtung Es werden einige Experten gefragt um die vorhandenen Ergebnisse zu sichten und zu kommentieren bzw. zu bewerten. Da dieser Prozess subjektiver Natur ist, gibt es hierbei lediglich die Möglichkeit, den Durchschnitt der Bewertungen zu betrachten. Auch wenn diese Methode eher skurril erscheinen mag, so wird sie dennoch häufig angewandt.
Diese Art der Evaluation ist selbstverständlich sehr zeit- und arbeitsintensiv. Jedoch gibt es manchmal Situationen in denen es keine andere Möglichkeit gibt, da keine Methoden vorhanden sind, welche die Qualität der Ergebnisse garantieren könnten. Es sollte jedoch auch angemerkt werden, dass diese Möglichkeit Daten/Cluster zu evaluieren stark von den Daten selbst abhängt. Während es bei Textdokumenten noch möglich erscheint, da diese gelesen und bewertet werden können, macht es bei einer Vielzahl von Zahlen, welche aus einer Datenbank stammen, wenig Sinn.

Ground Truth Bei dieser Methode nimmt man die oben erwähnten Klassifikationsmengen zu Hilfe. Man geht hierbei von der Annahme aus, dass jeder Cluster einer Klasse der Klassifikationsmenge entspricht. Bei vier verschiedenen Klassen würde man also auch den Clusteringalgorithmus vier Cluster erzeugen lassen. Im nächsten Schritt wird geschaut ob die Instanzen in den jeweiligen Clustern die gleichen Klassen beim Klassifikationsalgorithmus erhalten haben. Dabei können diverse Metriken genutzt werden, um beide Verfahren gegeneinander zu vergleichen: **Entropy** und **purity** sind hier zwei neue Maße, während precision, recall und F-Score bereits aus dem Bereich der Klassifikationsprobleme bekannt sind.
Auch in diesem Fall kann eine Konfusionsmatrix erstellt werden, um daraus diverse Werte ableiten zu können. Dabei gilt im Folgenden: $C = (c_1, c_2, ..., c_k)$ die Menge an verschiedenen Klassen im Datensatz D. Somit erstellt auch das Clustering k verschiedene Cluster, welches D in k disjunkte Teilmengen aufteilt.

Entropie kann für jeden Cluster wie folgt berechnet werden:

$$entropy(D_i) = -\sum_{j=1}^{k} Pr_i(c_j log_2 Pr_i(cj)) \tag{10}$$

Dabei stellt $Pr_i(c_j)$ die Proportion der Datenpunkte im Cluster i der Klasse c_j dar.

Die Entropie für alle Cluster kann wie folgt berechnet werden:

$$entropy_{total}(D) = \sum_{i=1}^{k} \frac{|D_i|}{|D|} \times entropy(D_i) \tag{11}$$

Purity Hierbei handelt es sich um den Spezialfall, dass ein Cluster nur aus einer Klasse an Daten besteht. Der Vollständigkeit halber sind im Folgenden die beiden Formeln (12 und 13) zur Berechnung der Purity dargestellt. Im Anschluss folgt ein Beispiel, welches beides noch einmal anschaulich verdeutlicht.

$$purity(D_i) = \max_j(Pr_i(c_j)) \tag{12}$$

$$purity_{total}(D) = \sum_{i=1}^{k} \frac{|D_i|}{|D|} \times purity(D_i) \tag{13}$$

Bei dem folgenden Beispiel in Abbildung 46. geht man von einer Datenmenge von 900 Textdokumenten aus, welche sich in drei verschiedene Themen gliedern lassen. (Science, Sports, Politics). Jede Klasse besteht aus 300 Dokumenten wobei jedes der Dokumente zu einer der drei erwähnten Klassen gehört. Es wird ein Clustering auf dieser Datenmenge durchgeführt, wobei in diesem Fall keine Klassen bzw. Labels vorhanden sind. Nachdem dies geschehen ist, möchte man eine Messung durchführen, die eine Aussage über die Effektivität des Algorithmus trifft.
Im ersten Schritt wird, wie im Klassifikationsfall, eine Konfusionsmatrix aus den berechneten Daten angelegt. Wie man sieht hat der erste Cluster 250 wissenschaftliche und 20 sportbezogene Dokumente, sowie 10 Dokumente mit politischem Inhalt erhalten. Die anderen beiden Cluster sind ähnlich aufgebaut. Man kann an den beiden letzten Spalten (Entropy und Purity) sehen, dass der erste Cluster deutlich besser als die anderen beiden ist. Zusätzlich kann man die gesamte Entropie sowie Purity nehmen, um denselben Algorithmus mit verschiedenen Parametereinstellungen zu vergleichen. Werte wie precision, recall oder F-Score können auf ähnliche Weise berechnet werden, wie dies schon bei der Klassifikation der Fall war.

Anmerkung: Auch wenn ein Algorithmus gute Ergebnisse bei gelabelten Daten anzeigt, so gibt es **keine** Garantie, dass dies auch bei Daten der Fall ist, bei denen es keine Labels gibt.

Cluster	Science	Sports	Politics	Entropy	Purity
1	250	20	10	0.589	0.893
2	20	180	80	1.198	0.643
3	30	100	210	1.257	0.617
Total	300	300	300	1.031	0.711

Abbildung 46: Konfusionsmatrix für Clusteringevaluation

5.5 Abbildung auf das Framework

5.5.1 Einleitung

Nachdem die diversen Evaluationsprobleme nun aus theoretischer Sicht betrachtet wurden, folgt in den nächsten Abschnitten eine Erläuterung der Umsetzung innerhalb des Frameworks. Es werden jeweils die wichtigsten Prinzipien beleuchtet - für detailliertere Betrachtungen sei an dieser Stelle auf die JavaDoc verwiesen.

5.5.2 Evaluation der Modellgüte

Die Evaluation der Güte eines Modells findet innerhalb des Frameworks mit Hilfe eines *QualityCalculator*-Knotens statt. Die grundsätzliche Funktionsweise ist für Klassifikationsprobleme und Regressionsprobleme identisch - lediglich die Ansteuerung des Knotens unterscheidet sich in den jeweiligen Problemklassen. Die Arbeitsweise des Knotens wird im Folgenden anhand der Abbildung 47 erläutert. Der Knoten selbst ist das einzige im Knotennetzwerk ansteuerbare Element des Aufbaus. Er kann über Events ein (im Fall von Klassifikationsproblemen) oder mehrere (im Fall von Regressionsproblemen) Modell(e) erhalten, welche zentraler Gegenstand der Güteevaluation sind. Abhängig vom Modelltyp - hier wird im Framework nach Prediction, Description und SelectiveDescription unterschieden - muss bei der Erzeugung des Knotens eine andere Implementierung des QualityCalculators instanziiert werden. Die Ansteuerung innerhalb des QualityCalculator-Knotens geschieht jedoch über das Interface QualityCalculator und verursacht somit keine Fallunterscheidungen im weiteren Quellcode des Knotens.
Im Rahmen der Evaluation der Modellgüte können verschiedene Gütemaße verwendet werden, welche ebenfalls bei der Erzeugung des QualityCalculator-Knotens instanziiert werden und zuvor über einen Knotenparameter konfiguriert werden müssen. Alle Umsetzungen von Gütemaßen müssen mindestens eines der Interfaces *PredictionQualityCriterion*, *DescriptionQualityCriterion* und *SelectiveDescriptionQualityCriterion* implementieren, welche allesamt vom allgemeineren Interface QualityCriteri-

on erben. Diese Unterscheidung nach den drei Modelltypen ist notwendig, da alle Modelltypen unterschiedliche Methoden zur Ergebnisabfrage bieten und somit individuell angesteuert werden müssen. In der Abbildung findet sich als Beispielimplementierung das Gütemaß „MeanSquaredError (MSE)“. Dieses hat als einzige Anforderung, zwei Modelle eines beliebigen aber festen Typs zu erhalten, sowie im Falle von PredictionModels und SelectiveDescriptionModels eine Liste von Parametern, die in die predict(..) bzw. describe(..) Methode der jeweiligen Modelle gegeben werden können, um die miteinander zu vergleichenden Werte zusammenzustellen.
Da über den Knoten eine beliebig große Menge an Modellen empfangen werden kann und somit die gleichzeitige Evaluation mehrerer Verfahren möglich ist, wurden Strategien implementiert, wie die Modelle einer Modellmenge für die Güteevaluation verwendet werden sollen. So ist es möglich, das gewählte Gütemaß auf jedes Modell einzeln anzuwenden, alle Modelle gleichzeitig in einem Gütemaß zu evaluieren, oder jedes Modell mit dem explizit zu kennzeichnenden, die Wahrheit repräsentierenden Modell zu evaluieren. Hierbei muss das gewählte Gütemaß die entsprechende Strategie natürlich unterstützen. Im Beispiel von MeanSquaredError kann lediglich eine paarweise Evaluation erfolgen, d.h. es wird immer ein Modell zusammen mit dem Wahrheitsmodell in das Gütemaß gegeben. Die Identifizierung des Wahrheits-Modells geschieht dabei implizit über die im Knoten konfigurierbare Reihenfolge der Modelle - das erste Modell in der spezifizierten Reihenfolge wird als Wahrheit behandelt - und kann innerhalb des Gütemaßes als gegeben angesehen werden.
Die Details der Gütemaße sind allesamt für den QualityCalculator-Knoten nicht relevant, so dass eine problemlose Erweiterbarkeit durch weitere Gütemaße gegeben ist. Die beschriebenen für Klassifikationsprobleme nötigen Gütemaße wie Precision, Recall, usw. wurden ebenfalls über diese Struktur implementiert, was zur Folge hat, dass der Knoten selbst die Güteevaluation von Klassifikationsproblemen und Regressionsproblemen identisch handhaben kann.

Je nachdem, ob ein Klassifikationsproblem oder ein Regressionsproblem betrachtet wird, ist eine andere Integration in das Knotennetzwerk erforderlich. Diese wird im Folgenden beschrieben.

Klassifikationsprobleme erfordern die Abbildung der in Kapitel 5.2 dargelegten Strategien der Aufteilung von Daten in Trainings- und Testdaten. Dies geschieht über das recht komplex zu konfigurierende Zusammenfügen von elementaren Knoten wie beispielsweise Sampler, die Events nach einer bestimmten Verteilungsrate auf unterschiedliche Nachfolger aufteilen oder Puffer, die bestimmte Daten aus Events zu einer Ansammlung zusammenfügen und diese auf Kommando oder beim Erreichen einer bestimmten Anzahl an Elementen gesammelt ausgeben. Da sich diese Herangehensweise als ebenso flexibel wie konfigurationsaufwändig herausstellte, wurden Klassen bereitgestellt, um die Konfiguration auf ein Minimum zu beschränken und trotzdem die gängigsten Konstellationen abzubilden. Für die schematischen Zusammenstellungen der Knoten zu einer solchen Datenaufteilungsstrategie im Klassifi-

kationskontext, wie sie innerhalb der Hilfsklassen abgebildet wurde, sei erneut auf das Kapitel 5.2 verwiesen. Allen Aufbauten ist jedoch gemein, dass sie (klassifizierte) Beispiele an den QualityCalculator-Knoten senden, welche dann zusammen mit dem jeweils aktuellen Modell in eine oder mehrere Klassifikations-Gütemaße gegeben werden. Intern erfolgt dort eine Einsortierung in eine *Konfusionsmatrix*, aus der dann die relevanten statistischen Daten gewonnen werden.

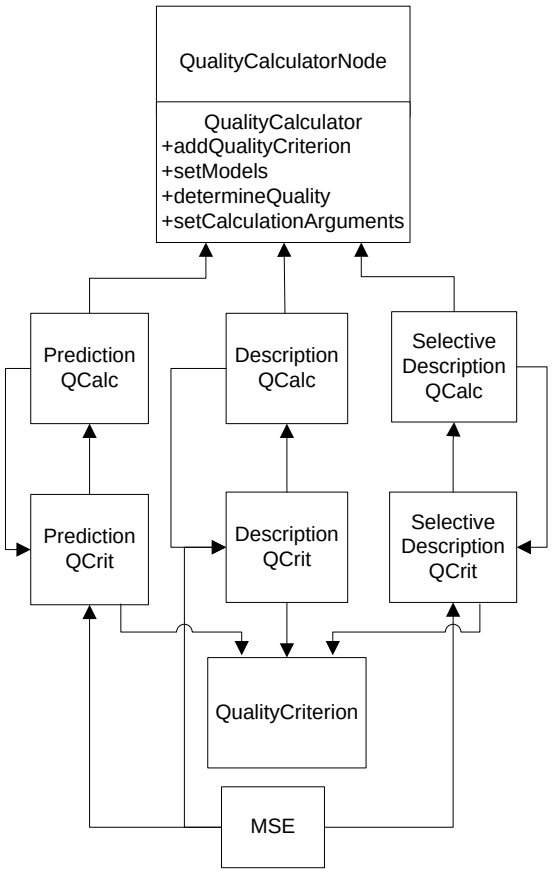


Abbildung 47: QualityCalculator-Struktur

Regressionsprobleme sind in der Konfiguration deutlich weniger komplex als die Klassifikationsprobleme. Die Abbildung auf das Framework entspricht der schematischen Darstellung in 48.

Es werden ein die *Wahrheit* abbildendes Lernverfahren und beliebig viele zu evaluierende Verfahren durch eine gemeinsame Datenquelle beliefert. Die hieraus entstehenden Modelle werden in einen Modellaggregator gegeben, der - anhand einer gemeinsamen Id - zusammengehörige Modelle identifiziert und als Modellmenge an den QualityCalculator-Knoten weiterreicht. Dort werden alle konfigurierten Gütemaße errechnet und die Ergebnisse ausgegeben.

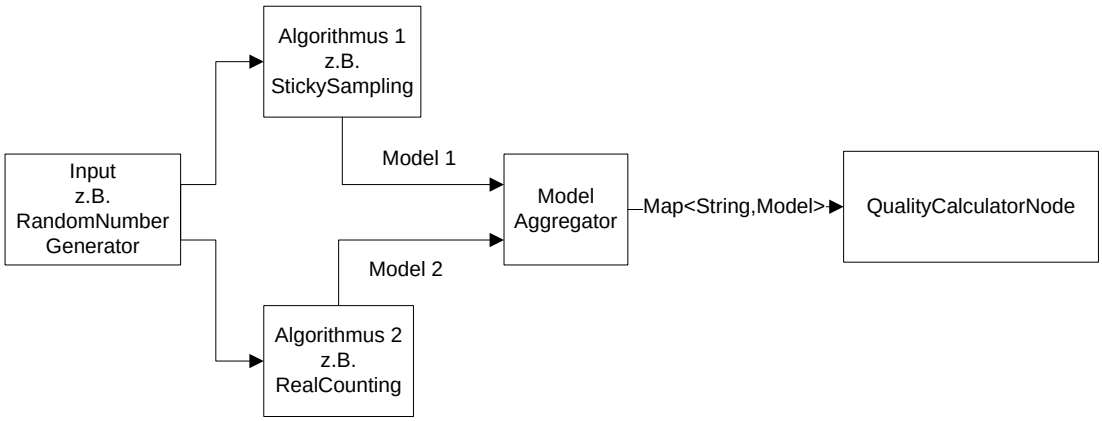


Abbildung 48: Modellgüte Regressionsprobleme

5.5.3 Zeitevaluation

Durch einen sogenannten *NodeWrapper* der Zeitevaluation wird für jeden Lerner-Knoten für jedes eingehende Event vor und nach der Ausführung des Lernschrittes die aktuelle Zeit in Millisekunden festgehalten. Die Differenz hieraus wird in einer Liste abgelegt, welche wiederum in regelmäßigen Abständen an einen Knoten im Netzwerk weitergeleitet und dort als durchschnittliche Verarbeitungsdauer des letzten Abfrageintervalls aufbereitet wird. Für die Kapselung der Lerner-Knoten in die NodeWrapper ist der NodeContainerHook *TimeEvaluationHook* erforderlich.

5.5.4 Durchsatzevaluation

Ebenso wie bei der Zeitevaluation wird jeder Lerner-Knoten mit einem für die Durchsatzevaluation konzipierten NodeWrapper umhüllt, welcher pro Abfrageintervall die Anzahl der verarbeiteten Events zählt. Auch hier erfolgt die Aufbereitung der Daten in einem eigenständigen Knoten des Netzwerkes. Für die Kapselung in NodeWrapper wird der NodeContainerHook *ThroughputEvaluationHook* benötigt.

5.5.5 Speicherevaluation

Die Speicherevaluation erhält über einen NodeContainerHook Zugriff auf alle Knoten des Netzwerks. Via *reflection* werden die Referenzen aller *Learner*-Instanzen innerhalb der Knoten ausgelesen und - identifiziert durch die Id des Knotens - unmittelbar nach der Initialisierung des jeweiligen Knotennetzwerkes in den Speicherevaluations-Knoten gegeben. Dieser kann mit den relevanten Knoten-Ids konfiguriert werden.

Damit dem Speicherevaluations-Knoten die Learner-Instanzen zur Verfügung stehen, ist es zwingend erforderlich, als NodeContainerHook den *SizeEvaluationHook* zum Container hinzuzufügen. Die Integration in das Knotennetzwerk erfolgt lediglich über die Anbindung eines geeigneten Knotens zur Ausgabe der Messresultate.
Die Speichermessung erfolgt dann automatisiert in regelmäßigen Abständen, wobei die Größe des jeweiligen Lerners und seines aktuellen Modells gemessen werden. Die Messung selbst erfolgt standardmäßig über eine Methode aus dem Apache Wicket Projekt[1]. Hierbei wird ein Objekt serialisiert und die Größe der serialisierten Daten, in Bytes gemessen, zurückgegeben. Wenn ein Lerner oder ein Modell nichtserialisierbare Felder enthält, kann auf diese Weise keine Speichernutzung gemessen werden, weshalb eine Ausgabe von *Double.NaN* erfolgt, was eine Verrechnung mit anderen Ergebnissen unmöglich macht. Wenn seitens des Entwicklers keine Möglichkeit besteht, die betroffenen Objekte serialisierbar zu machen, kann das jeweilige Objekt das Interface *Measurable* implementieren, welches eine Methode *getBytes()* enthält, die bei der Speichermessung immer bevorzugt behandelt wird.

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

6 Implementierte generische Verfahren

6.1 Zählalgorithmen

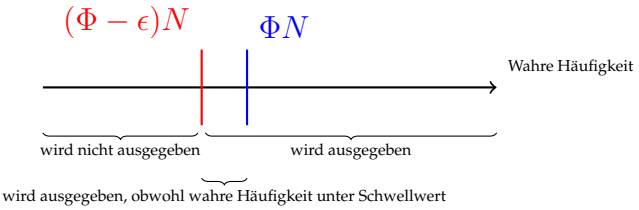
Zählalgorithmen sind wichtige generische Verfahren, da sie häufig die Basis für viele weitere Lernverfahren bilden. Sie sind auch gut dazu geeignet die Problematik bei Streamverfahren zu verdeutlichen. Die Aufgabenstellung ist klar und gut verständlich, wie auch die Restriktion im Speicherverbrauch.

Im folgenden Kapitel werden die Zählalgorithmen vorgestellt, die in der Seminarphase vorgestellt und dann im Laufe der Projektgruppe implementiert wurden.

6.1.1 Lossy Counting

Idee Zählen ist simpel, wenn man genug Platz hat. Wie zählt man jedoch, wenn man nicht zum Datenstrom linear viel Speicher benutzen will bzw. kann? Durch Abschätzen! Der `Lossy Counting` Algorithmus [31] geht so vor und wird oft als Ausgangsbasis für weitere Algorithmen benutzt. Es werden dabei zwei Parameter benötigt. Zum einen ein Schwellwert $\Phi \in (0, 1)$ für die erforderliche Häufigkeit eines Elementes im Datenstrom, damit dieser überhaupt ausgegeben wird und einen maximal erlaubten Fehler $\epsilon \ll \Phi$. Mit diesen beiden Parametern macht der Algorithmus folgende Garantien für das Zählen auf einem Datenstrom (wobei N die Anzahl der bisherigen Elemente im Datenstrom ist):

1. **Alle** Elemente mit der absoluten Vorkommen von echt größer ΦN werden **ausgegeben**
2. **Kein** Element mit dem absoluten Vorkommen von echt kleiner $(\Phi - \epsilon)N$ wird **ausgegeben**
3. Die geschätzte Häufigkeit ist dabei stets höchstens ϵN kleiner als die tatsächlich Häufigkeit



Solche Annahmen sind typisch für diese Art von Algorithmen und für viele Anwendungen auch absolut ausreichend. Der Gewinn durch diese Einschränkungen ist dabei ein logarithmischer Platzbedarf.

Umsetzung `LossyCounting` unterteilt den Stream in gleich große Teile - sogenannte `Buckets`. Wie groß diese sind, hängt von dem maximal erlaubten Fehler ab und beträgt $\frac{1}{\epsilon}$.

Eingehende Elemente werden in einer Datestruktur gespeichert, in der neben dem Element auch seine Häufigkeit und das `Bucket` in dem das Element eingefügt wurde, gespeichert werden. Am Ende eines jeden `Buckets` wird jedes Element in der Datenstruktur überprüft, ob es gelöscht werden kann. Nur die Elemente verbleiben in der Datenstruktur, die in jedem `Bucket` seit ihrem ersten Einfügen mindestens einmal aufgetaucht sind. Ist dies nicht der Fall, können wir das Element aus der Datenstruktur löschen und haben dadurch, dass wir die `Bucket`-Größe anhand des maximal erlaubten Fehlers bestimmt haben, einen Fehler von maximal ϵ gemacht.

Der `Lossy Counting` Algorithmus kann dabei auch mühelos auf häufige Mengen umgestellt werden. Dabei wird für jede Transaktion im Datenstrom die Potenzmenge gebildet. Diese Teilmengen können nun als normale Eingabe für den Algorithmus benutzt werden. Durch einige weitere Modifikationen wird dieser naive Ansatz auch effizienter beim Platzbedarf.

Knotenklasse

edu.udo.cs.pg542.util.node.learner.counter.LossyCountingNode

Eingaben

Schlüssel	Typ	Bedeutung
event:learner:input	siehe Parameter	Ein Element des Universums, über dessen Werte gezählt wird. Der Typ für eine laufende Instanz ist dabei beliebig aber fest und wird über den Parameter 'input-class' gesetzt.

Ausgaben

Schlüssel	Typ	Bedeutung
event:learner:output	LossyCounting-Model	Für eine Mindesthäufigkeit werden alle Elemente ausgeben, die mit einem Fehler von 'maxError' noch häufig sind.

Parameter

Schlüssel	Typ	Opt.	Bedeutung
input-class	Class	Nein	Die Eingabeklasse für die LossyCounting Instanz.
max-error	Integer	Nein	Der maximale Fehler der bei der Abschätzung der Häufigkeiten erlaubt ist.

6.1.2 Sticky Sampling

Sticky Sampling ist ein probabilistisches Zählverfahren, das der approximativen Findung häufiger Elemente eines Datenstroms dient. Dabei wird ein Element als *häufig* bezeichnet, wenn seine geschätzte Anzahl an Vorkommen im bisher betrachteten Teil des Datenstroms einen benutzerdefinierten Schwellwert überschreitet.

Im Folgenden sei N die bisherige Länge des betrachteten Datenstroms, $s \in (0, 1)$ der besagte Häufigkeitsschwellwert und $\epsilon \in (0, 1)$ eine Fehlerschranke. Dabei sei $\epsilon \ll s$.

Zu jedem beliebigen Zeitpunkt soll eine Liste der häufigen Elemente, zusammen mit ihren geschätzten Zählwerten abgefragt werden können. Dazu wird als Datenstruktur S eine Menge von Tupeln (e, f) verwaltet, wobei e ein Element des Datenstroms darstellt, während f die zugehörige approximierte Anzahl an Vorkommen im Datenstrom repräsentiert. Da das Verfahren den Anspruch stellt, speichereffizient zu sein, ist es nicht möglich, für jedes jemals im Datenstrom vorkommende Element ein solches Tupel zu verwalten - es wird eine Strategie benötigt, um nur einen Teil der Elemente in die Datenstruktur aufzunehmen oder sogar wieder zu entfernen. Dabei sollen für die Ergebnisabfrage folgende Eigenschaften erfüllt werden:

1. Real häufige Elemente, also Elemente, deren Anzahl an Vorkommen im Datenstrom mindestens sN beträgt, werden immer als häufig erkannt
2. Elemente die nicht häufig sind können nur mit Zählwerten zwischen $(s - \epsilon)N$ und sN fälschlicherweise als häufig erkannt werden
3. Die geschätzte Häufigkeit eines Elementes ist höchstens um ϵN kleiner als die reale Häufigkeit des Elementes

Über die oben genannten konfigurierbaren Parameter s und ϵ hinaus gibt es eine Fehlschlagwahrscheinlichkeit $\delta \in (0, 1)$, die bestimmt, wie häufig gegen eine oder mehrere der soeben eingeführten Eigenschaften verstoßen werden darf.

Die Entscheidung, ob ein Element e, das noch nicht in S verwaltet wird zu S hinzugefügt werden soll, ist abhängig von der Sampling-Rate r und geschieht mit der

Wahrscheinlichkeit $\frac{1}{r}$. Dabei wächst die Anzahl der aufeinanderfolgenden Elemente, für die mit dem selben Wert von r über das Hinzufügen entschiedenen wird mit der Länge des Datenstroms: Sei $t = \frac{1}{\epsilon}\log(s^{-1}\delta^{-1})$. Dann ist für die ersten $2t$ Elemente $r = 1$, für die nächsten $2t$ Elemente $r = 2$, für die nächsten $4t$ Elemente $r = 4$ und so fort.

Vor der Betrachtung des ersten Elementes des Datenstrom ist $S = \emptyset$ und $r = 1$. Danach wird für jedes eingehende Element e geprüft, ob es bereits mit einem Tupel (e, f) in S vertreten ist. Ist dies der Fall, wird der zugehörige Zählwert f um eins erhöht. Ansonsten wird es als $(e, 1)$ mit Wahrscheinlichkeit $\frac{1}{r}$ zu S hinzugefügt.

Jedesmal wenn r sich ändert, wird ein Vorgang ausgeführt, der den Sinn hat, die Elemente in S so umzuformen, als ob sie mit der neuen Sampling-Rate ausgewählt worden wären. Dazu wird für jedes Tupel in S solange ein Münzwurf simuliert, bis ein vorher als *Misserfolg* festgelegtes Ergebnis des Münzwurfes eintritt. Solange wird pro erfolgreichem Münzwurf f um eins dekrementiert. Wird auf diesem Wege der Zählwert 0 erreicht, so wird das Tupel $(e, 0)$ aus S entfernt - mit der neuen Sampling-Rate wäre dieses Element nie für S ausgewählt worden.

Eine Abfrage der häufigen Elemente aus der so verwalteten Datenstruktur S hat als Ergebnis alle Tupel (e, f), für die $f \geq (s - \epsilon)N$ gilt.

Knotenklasse

edu.udo.cs.pg542.util.node.learner.counter.StickySamplingNode

Eingaben

Schlüssel	Typ	Bedeutung
event:learner:input	siehe Parameter	Ein Element des Universums, über dessen Werte gezählt wird. Der Typ für eine laufende Instanz ist dabei beliebig aber fest und wird über den Parameter 'input-class' gesetzt.

Ausgaben

Schlüssel	Typ	Bedeutung
event:learner:output	StickySampling-Model	Das aktuelle Modell, aus dem eine Liste der häufigen Elemente angefragt werden kann. Ferner ist es möglich für ein Element des Universums, über dessen Werte gezählt wird, eine Anfrage zu stellen, was der aktuelle Zählwert dieses Elementes ist, sowie ob es häufig ist.

Parameter

Schlüssel	Typ	Opt.	Bedeutung
input-class	Class	Nein	Die Eingabeklasse für die LossyCounting Instanz.
min-support	Double	Nein	Der Schwellwert, dessen Überschreiten ein Element zu einem häufigen Element macht.
max-error	Double	Nein	Eine Abweichung des Schwellwertes nach unten. Im Bereich $minSupport - maxError$ wird ein Element auch dann als häufig angesehen, wenn es nicht häufig ist.
max-failure	Double	Nein	Siehe Punkt 3 der obigen Aufzählung.

6.1.3 Count(Min)Sketch

Idee `Count Sketch` [10] ist ein von Google entwickelter Top-k Algorithmus, der ursprünglich zur Bestimmung von Suchtrends (Google Zeitgeist) benutzt wurde. Obwohl der Algorithmus eigentlich zur Bestimmung der k häufigsten Elemente dient, werden zunächst die Häufigkeiten aller Elemente abgeschätzt, um daraus dann die k häufigsten Elemente zu ermitteln.

Bei dieser Abschätzung wird die Häufigkeit eines Elements als Zufallsvariable betrachtet, wobei der Erwartungswert dem tatsächlichen Vorkommen entspricht. `Count`

`Sketch` versucht für jedes Element so eine Zufallsvariable zur Verfügung zu stellen, wobei die Varianz möglichst klein sein soll.

Eine sehr einfache Zufallsvariable für Elemente in einem Datenstrom wäre ein Zähler für alle Elemente zu benutzen. Je mehr Elemente im Datenstrom auftauchen, desto größer ist die Varianz in dieser "Zufallsvariable". Das Ziel von `Count Sketch` ist es nun diese Varianz soweit zu minimieren, dass brauchbare Ergebnisse entstehen. Der erste Schritt hierbei ist es mehrere Zähler zu benutzen, wobei mit einer Hashfunktion bestimmt wird, welches Element welchen Zähler inkrementiert. Wichtig hierbei ist die Wahl der Hashfunktion. Für die Einhaltung der Fehlerschranken muss sie paarweise unabhängig sein.

Mit der Verwendung einer Hashfunktion und mehreren Zählern haben wir die Varianz der Häufigkeit schon deutlich verbessert. Um diese noch weiter zu verringern, können wir noch weitere, unterschiedliche Hashfunktionen mit dazu gehörigen Zählern erstellen, die für ein Element immer andere Zähler inkrementiert. Wenn wir bei der Abschätzung der Häufigkeit von den zum Element gehörenden Zähler den Durchschnitt bilden, ist die Varianz nochmal deutlich kleiner. Je mehr Hashfunktionen und Zähler wir dabei verwenden, desto genauer wird natürlich das Ergebnis, benötigt dann aber natürlich mehr Speicher.

Über dies hinaus verwendet `Count Sketch` für jedes Element noch eine weitere Hashfunktion, die ein Element auf $\{-1, +1\}$ abbildet. Dieser Wert wird bei jedem Zähler aufaddiert, so dass ein Element seinen Zähler beispielsweise inkrementiert, ein anderes Element, das auch auf diesen Zähler abgebildet wurde, den gleichen Zähler dekrementiert. Dahinter steht der Versucht zu verhindern, dass ein Zähler zu groß wird und die Abschätzung mehr dem wahren Wert entspricht.

Eine `Count Min Sketch` [14] genannte Variante verzichtet auf diese Hashfunktion, die entscheidet, ob ein Zähler in- oder dekrementiert wird und bildet über die verschiedenen Hashfunktionen und ihren Zählern nicht den Durchschnitt, sondern benutzt den kleinsten Wert als Abschätzung. Da wir auf jeden Zähler nur positive Werte addieren, können wir nie zu wenig zählen, sondern höchstens zu viel. Da die unterschiedlichen Hashfunktionen auf unterschiedliche Zähler abbilden, können wir dadurch, dass wir den kleinsten Wert wählen, den Fehler minimieren.

Umsetzung Die Implementierung beider Algorithmen beschränkt sich also hauptsächlich darauf gute Hashfunktionen zu entwickeln. In [9] wird beschrieben, wie wir solche paarweise unabhängigen Hashfunktionen erstellen können.

Knotenklasse

edu.udo.cs.pg542.util.node.learner.counter.CountSketchNode

Eingaben

Schlüssel	Typ	Bedeutung
event:learner:input	siehe Parameter	Ein Element des Universums, über dessen Werte gezählt wird. Der Typ für eine laufende Instanz ist dabei beliebig aber fest und wird über den Parameter 'input-class' gesetzt.

Ausgaben

Schlüssel	Typ	Bedeutung
event:learner:output	CountSketch-Model	Das Model für Count Sketch Algorithmen, das eine Häufigkeitsabschätzung für ein einzelnes Element zurück gibt, aber auch ein Top-k Modell.

Parameter

Schlüssel	Typ	Opt.	Bedeutung
sketch-type	String	Nein	Bestimmt ob enweder 'count-sketch' oder 'count-min-sketch' verwendet wird
k	Integer	Nein	Parameter für die Bestimmung der k häufigsten Mengen
domain	Integer	Nein	Erwartete Domäne der Anwendung, d.h. wieviele unterschiedliche Elemente werden in etwa im Stream erwartet.
hash-functions	Integer	Nein	Die Anzahl der Hashfunktionen, die erstellt werden, um den Durchschnitt respektive das Minimum für die Abschätzung zu wählen.
buckets	Integer	Nein	Anzahl der Buckets bzw. Zähler auf die eine einzelne Hashfunktion abbilden kann.

6.1.4 Space Saving

Idee Der hier vorstellte Space Saving Algorithmus ist aus [34] und dient wie die bisherigen Algorithmen in diesem Kapitel dazu, sowohl häufige Mengen, wie auch die Top-K Elemente eines Streams zu bestimmen.

Die Idee dabei ist, dass beide Verfahren in einem Algorithmus gekoppelt werden. Dabei ist dieses Verfahren exakt und effizient, wenn das betrachtete Alphabet klein gehalten wird. Falls die Größe steigt, also eher der realistischen Fall eintritt, ist die hier beschriebene Lösung Speichereffizient. Bei einer Standard Datenverteilung gibt dieses Verfahren bei Top-K Anfragen, eine Menge der k' Elemente zurück, wobei $k' \approx k$ ist. Dabei wird garantiert, dass dieses k' Elemente die Top k' Elemente sind.

Der Algorithmus ist dabei sehr intuitiv und überschaubar. Falls ein Element e aus dem Stream betrachtet wird, wird geschaut ob es sich bereits in der Datenstruktur befindet und gezählt wird. In diesem Fall wird der Zähler des entsprechenden Elements erhöht. Falls das Element noch nicht enthalten ist, wird geschaut welches das Element mit der kleinsten Anzahl an *hits* ist - e_m. Im nächsten Schritt wird e_m durch e ersetzt, der Zähler $count_m$ wird inkrementiert und ϵ_m, also der Fehler, wird auf den Wert des ersetzten Elements e_m gesetzt.

Wichtig bei der Implementierung ist eine Datenstruktur, die er erlaubt effizient die besagten Zähler zu verwalten ohne dabei die Reihenfolge der Elemente zu verändern. Aus diesem Grund wird hierbei die *Stream-Summary* Datenstruktur vorgeschlagen. Diese ist in der Abbildung 49 dargestellt.

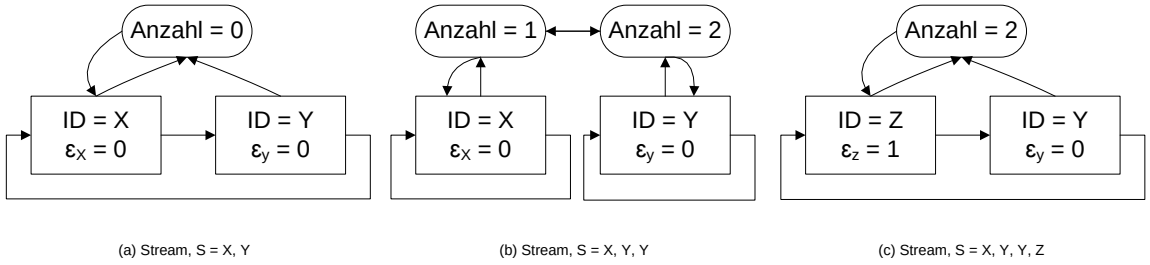


Abbildung 49: Stream Summary Datenstruktur

Umsetzung Die Implementierung des Algorithmen besteht hauptsächlich in der richtigen Wahl und Implementierung der Datenstruktur. Hierfür wurde, wie bereits erwähnt, die Stream-Summary Datenstruktur gewählt. Dabei muss auf darauf geachtet werden, dass sowohl nach dem Einfügen eines neuen Elements, wie auch beim Aktualisieren eines alten, jeweils die Datenstruktur sortiert wird. Durch diese Sortierung wird gewährleistet, dass die in [34] erwähnten Kriterien erfüllt werden.

Knotenklasse

edu.udo.cs.pg542.util.node.learner.counter.SpaceSavingNode

Eingaben

Schlüssel	Typ	Bedeutung
event:learner:input	siehe Parameter	Ein Element des Universums, über dessen Werte gezählt wird. Der Typ für eine laufende Instanz ist dabei beliebig aber fest und wird über den Parameter 'input-class' gesetzt.

Ausgaben

Schlüssel	Typ	Bedeutung
event:learner:output	SpaceSaving-Model	Das Model für den Space Saving Algorithmus, das eine Häufigkeitsabschätzung für ein einzelnes Element zurückgibt

Parameter

Schlüssel	Typ	Opt.	Bedeutung
input-class	Class	Nein	Die Eingabeklasse für die SpaceSaving Instanz.
counters	Integer	Nein	Die Anzahl der zur Verfügung stehenden Zähler
min-support	Double	Nein	Mindestsupport, ab dem ein Element als häufig gezählt wird,
max-error	Double	Nein	Der maximale Fehler der bei der Abschätzung der Häufigkeit erlaubt ist.

6.1.5 Evaluation: Lossy Counting, StickySampling und CountSketch

Die vorgestellten Zählalgorithmen sind allesamt Approximationsalgorithmen. Im Folgenden werden die verschiedenen Verfahren bezüglich Genauigkeit und Speicherverbrauch verglichen. Als Maß für die Genauigkeit wurde dazu der mittlere quadratische Fehler gewählt. Zur Evaluierung wurde ein Knoten entwickelt, der die Häufigkeiten exakt anhand einer *HashMap* berechnet.

Also Datenbasis wurden normalverteilte Zufallszahlen in einem Bereich zwischen 0 und $1.000.000$ gezogen.

Folgende Parameter wurden für die Zählalgorithmen eingestellt.

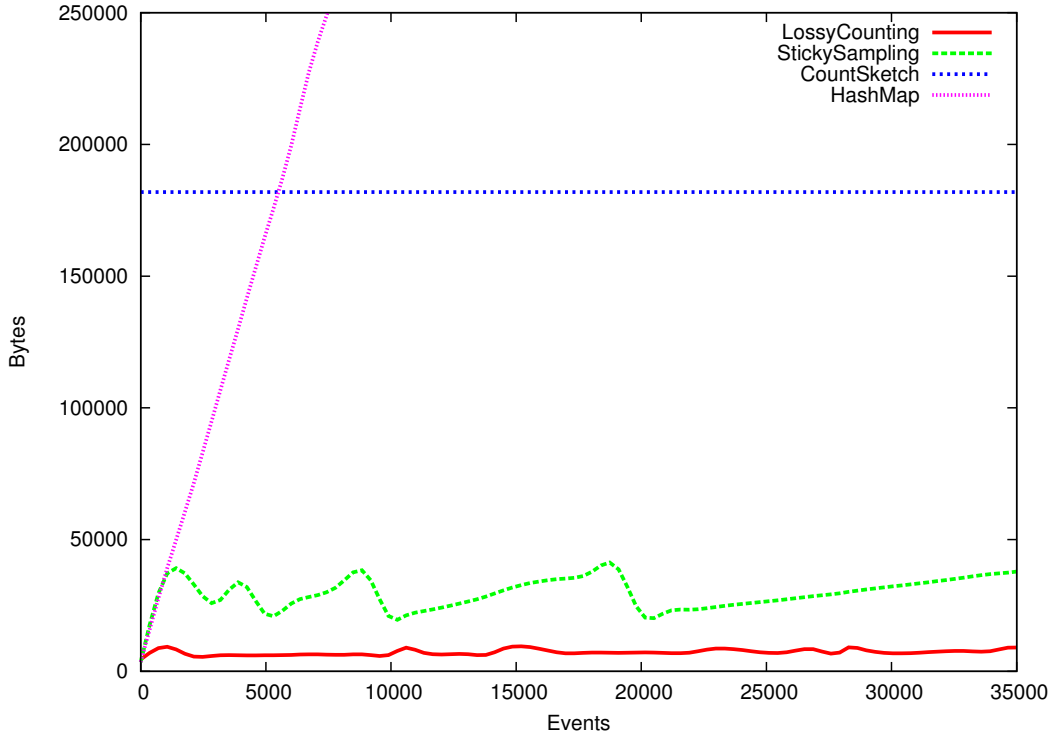
LossyCounting Für das ϵ wurde 0.01 angegeben. Eine Mindesthäufigkeit wurde dabei verzichtet, da wir anhand der Datenstruktur nur die Häufigkeiten approximiert haben.

StickySampling Eingestellt wurde auch hier ein ϵ von 0.01. Als δ wurde 0.02 gewählt. Auch hier haben wir nur Häufigkeiten approximiert und nicht die häufigen Elemente verglichen.

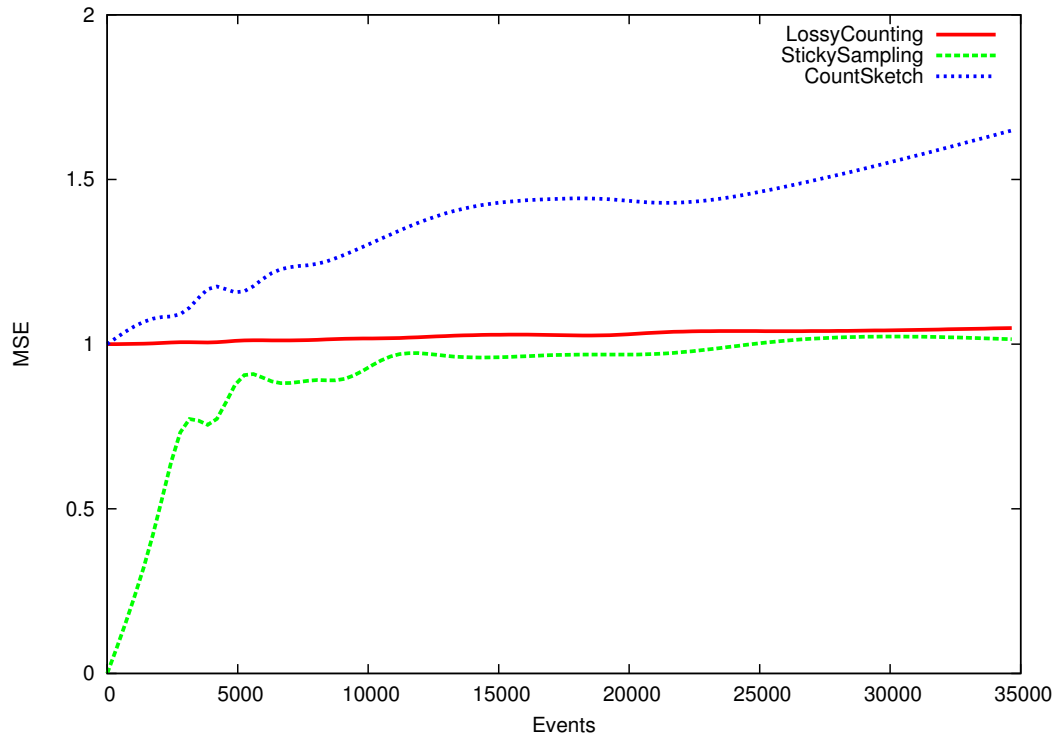
CountSketch Für den CountSketch wurden 300 2-unabhängige Hashfunktionen erstellt, die jeweils auf 300 Zähler zeigen.

Speicherverbrauch Der Speicherverbrauch der Knoten ist in Abbildung 50a zu erkennen. Der Knoten zur exakten Bestimmung der Häufigkeiten wächst erwartungsgemäß linear zum Datenstrom. Da CountSketch zu Anfang seine Datenstruktur mit Hashfunktionen und Arrays aufbaut und beibehält, ist hier der Speicher konstant und zu Anfang am höchsten. Am wenigstens Speicher benötigte StickySampling, gefolgt von LossyCounting, bei dem man die Komprimierungsphasen gut erkennen kann.

Mittlerer quadratischer Fehler (MSE) Um die Zählalgorithmen qualitativ zu bewerten, wurde der mittlere quadratische Fehler gegenüber der wahren Häufigkeit gemessen. Dies ist ein häufig benutztes Maß, das große Fehler stärker bestraft. Der gemessene Fehler von LossyCounting und StickySampling gleich sich im Laufe des Streams an und bleibt relativ konstant. CountSketch dagegen hat einen leicht höheren Fehler und steigt mit fortschreitendem Stream. Somit ist CountSketch qualitativ schlechter als die anderen beiden Algorithmen. Zu beachten ist hierbei aber, dass CountSketch ursprünglich als Top-k Algorithmus gedacht war. Die wirklich häufigen Elemente beeinflussen die von der Hashfunktionen zugewiesenen Zähler sehr stark, so daß Top-k Elemente von diesem Fehler nicht stark beeinflusst werden.



(a) Speicherverbrauch der Zählalgorithmen

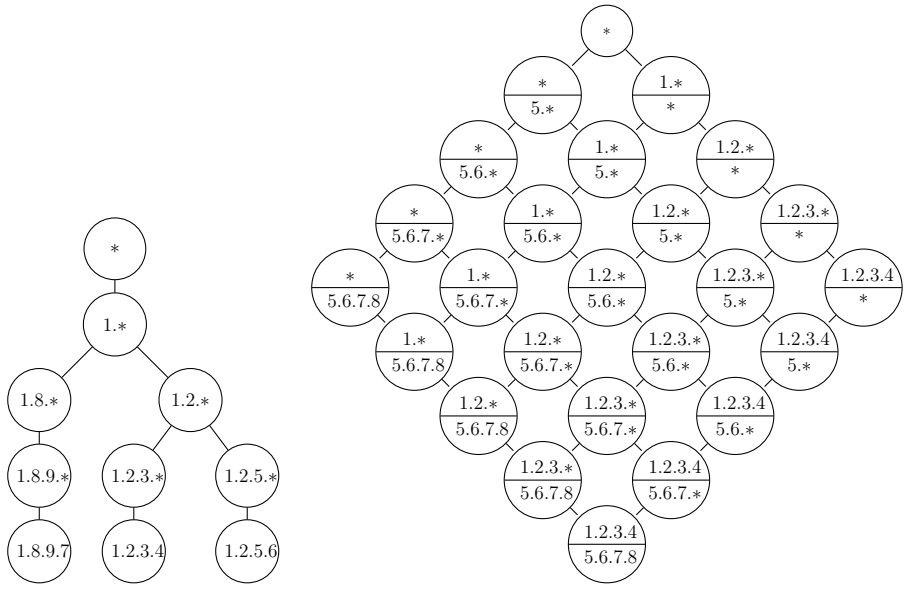


(b) Mittlerer quadratischer Fehler der Zählalgorithmen

Abbildung 50: Evaluierung von Lossy Counting, Sticky Sampling und CountSketch auf 35.000 normalverteilten Beispielen.

6.1.6 Hierarchical Heavy Hitters

Idee Ein *Heavy Hitter* ist ein Begriff der nichts anderes aussagt als: Dies ist ein häufiges Element! Im Grunde genommen also genau das, was der Lossy Counting Algorithmus zu seiner Mindesthäufigkeit Φ findet. Der entscheidende Punkt ist das Adjektiv Hierarchical. Wie schon kurz bei der Anwendung erwähnt, kann es vorteilhaft sein, nicht nur über ein Element selbst, sondern auch was über seine hierarchischen Teilstrukturen auszusagen. Wir wollen also auch Angriffe aus Teilnetzen zählen und nicht nur einzelne IP-Adressen.
Das naive Zählen in allen Hierarchiestufen würde uns dabei aber nicht weiterhelfen. Wir hätten keine Informationen darüber, wieso etwas häufig ist. In unserem konkreten Fall mit den IP-Adressen wäre das schlecht. Jedes Teilnetz wäre sofort gefährlich, wenn es einen Angreifer aus diesem Teilnetz gibt. Alle Kunden eines Providers auszusperren, weil ein Kunde einen Server angreift, wäre nicht optimal.
Insofern brauchen wir differenziertere Methoden. Für die Hierarchical Heavy Hitters (HHHs) wird folglich folgende Bedingung für das Zählen eingeführt: Nur konkrete Elemente einer Hierarchiestufe werden gezählt, die weder selbst häufig sind, noch zu einer anderen Hierarchiestufe weiter unten, die häufig ist, beitragen.



(a) Eindimensionale Hierarchie

(b) Mehrdimensionale Hierarchie

Abbildung 51: Ein- und mehrdimensionale Hierarchien auf IP-Adressen

Komplexer wird das Vorgehen, wenn man zusätzlich noch mehrere Dimensionen beachten muss - also auch noch die Empfängeradresse wichtig ist. Das bedeutet, dass Elemente nicht nur zu einer Hierarchiestufe zählen, was zusätzliche Probleme des Zuvielzählens hervorruft.

Umsetzung Der Algorithmus zur Berechnung von Hierarchical Heavy Hitters [13] ähnelt dem Algorithmus von LossyCounting (siehe Kapitel 6.1.1). Jedes Element wird in eine Datenstruktur eingefügt, die nach einer gewissen Anzahl von Elementen - abhängig von der angegebenen Fehlerschranke - regelmäßig komprimiert wird. Wie dies passiert, hängt von der gewählten Strategie ab. Diese Strategien unterscheiden sich im Speicheraufwand und Einfügegeschwindigkeit, aber auch in der Qualität der Lösung. Diese Qualität wird anhand der Anzahl der gefundenen Hierarchical Heavy Hitters gemessen. Mengen, die genau den Hierarchical Heavy Hitters entsprechen, sind optimal. Manche Strategien finden aber darüber hinaus noch weitere Elemente, die laut der Definition keine Hierarchical Heavy Hitters sind. Die Qualität einer solchen Lösung ist dann schlechter.

Folgende Strategien wurden implementiert:

NoStream Berechnet genau die Menge der Hierarchical Heavy Hitters, benötigt aber linearen Speicher und ist somit kein Stream-Algorithmus. Diese Implementation wird benutzt, um die Qualität und den Speicherverbrauch der Stream-Strategien zu evaluieren.

Naive Eine naive Herangehensweise wäre für jede auftauchende Hierarchiestufe einen Lossy Counting Algorithmus laufen zu lassen. Für jedes Element zählen dann die in der Hierarchie passenden Algorithmen mit. Die Qualität der Lösung ist dann aber nicht befriedigend. Es werden alle häufigen Elemente mit ihren Hierarchien gezählt.

FullAncestry Zwischen den einzelnen Hierarchiestufen werden - im Gegensatz zur naiven Strategie - Informationen ausgetauscht, um die exakten Hierarchical Heavy Hitters zu berechnen. Die Datenstruktur beinhaltet die Grenze im Hierarchiebaum, ab der Hierarchical Heavy Hitters geschätzt werden. Die Elemente in den Hierarchiestufen darüber werden ebenfalls gespeichert, um durch diese Zusatzinformationen beim Komprimieren der Datenstruktur nicht zu viele Informationen zu verlieren. Deswegen heißt diese Strategie auch `FullAncestry`-Strategie.

PartialAncestry Die `PartialAncestry`-Strategie versucht möglichst wenig weitere Elemente in den Hierarchiestufen zu speichern, um den Speicherverbrauch zu verringern. Da dadurch zusätzliche Informationen gelöscht werden, ist es auch schwieriger den Hierarchiebaum zu komprimieren, weshalb der Baum unter Umständen nicht weniger Speicher als `FullAncestry` benötigt. Die Evaluation hat aber ergeben, dass mit einem Speichergewinn zu rechnen ist.

Knotenklasse

edu.udo.cs.pg542.util.node.learner.counter.HierarchicalHeavyHitterNode

Eingaben

Schlüssel	Typ	Bedeutung
event:learner:input	Generalizable	Eingabeklasse muss die Schnittstelle Generalizable implementieren

Ausgaben

Schlüssel	Typ	Bedeutung
event:learner:output	Selective-Description-Model	Modell das für ein Threshold die berechneten Hierarchical Heavy Hitters ausgibt

Parameter

Schlüssel	Typ	Opt.	Bedeutung
strategy	Class	Nein	Die Strategie, die zur Berechnung der HHHs verwendet wird.
input-class	Class	Nein	Die Eingabeklasse für den HHH Algorithmus - muss die Generalizable Schnittstelle implementieren
epsilon	double	Nein	Der maximale Fehler bei der Berechnung der HHHs

6.1.7 Evaluation: Hierarchical Heavy Hitters

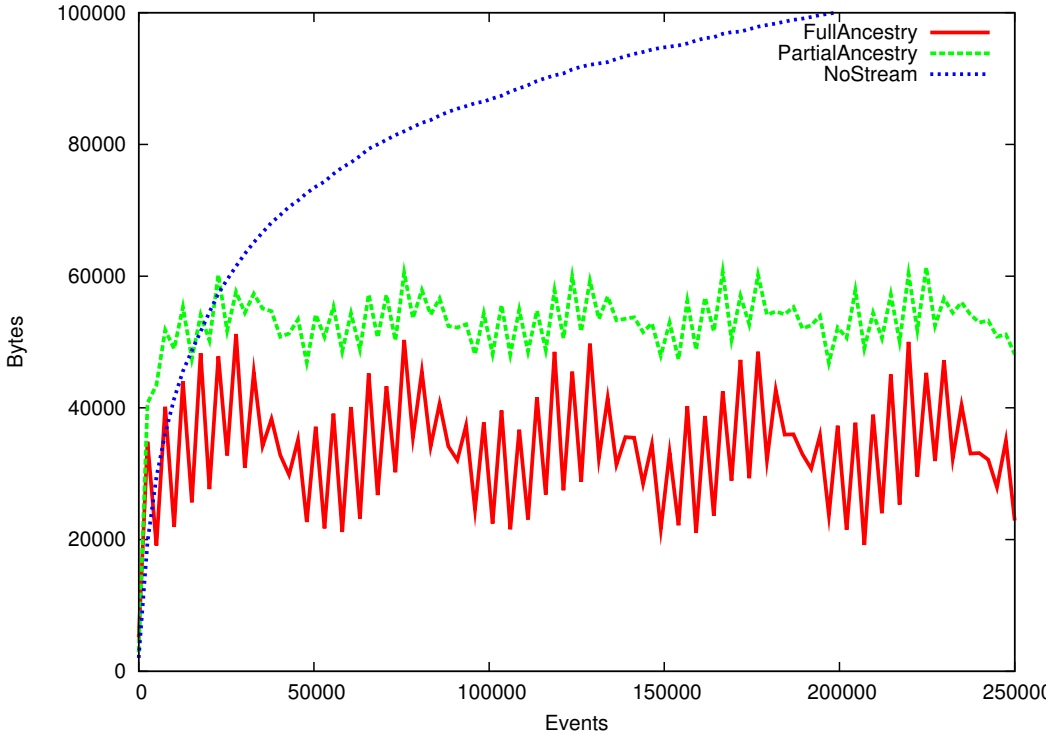
Wie bei den Zählverfahren handelt es bei den HHHs um einen approximativen Algorithmus. Auch hier ist der Speicherverbrauch - wie bei jedem Streamalgorithmus - entscheidend. Als Güte dahingegen verwenden wir die Anzahl der gefundenen HHHs. Je weniger HHHs gefunden werden, desto besser ist das Ergebnis. Die exakte Menge an HHHs wurde durch einen nicht Stream-fähigen Algorithmus bestimmt. (Siehe Kapitel 6.1.6)

Als Datenbasis zur Evaluierung wurden IP-Adressen aus einer Log-Datei[2] extrahiert. Als Mindesthäufigkeit wurde bei allen Strategien 0.002 eingestellt. Bei Partial- und FullAncestry wurde zusätzlich ein maximaler Fehler von $\epsilon = 0.001$ erlaubt.

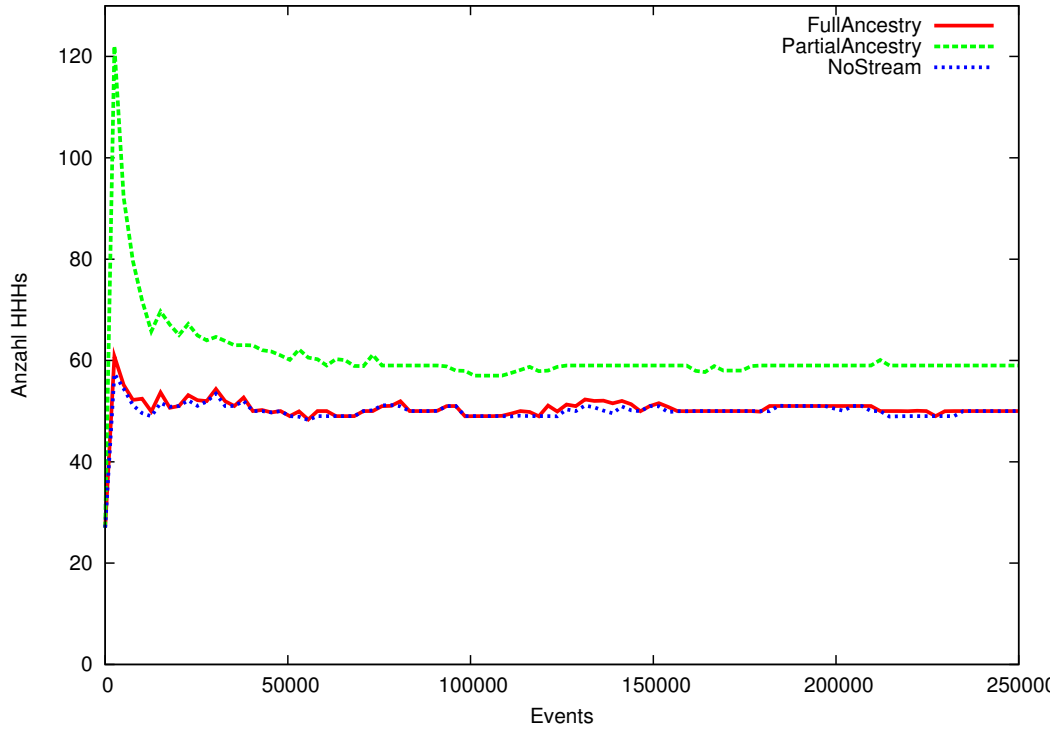
Speicherverbrauch Beide Stream-Strategien verbrauchen zu Beginn deutlich mehr Speicher als der Nicht-Stream Algorithmus (siehe Abbildung 52a). Erst ab 50.000 Adressen wird der Unterschied deutlich. Der Speicherverbrauch bleibt bei Partial- und FullAncestry zwischen 20 und 60 kb, während der exakte Algorithmus konstant steigt. Das starke Schwanken der Stream-Algorithmen zwischen 20 und 60 kb ist durch das stetige Komprimieren zu erklären.

Güte der Lösung Bei der Güte der Strategien ist die Anzahl der gefundenen HHHs entscheidend. Der Nicht-Stream-Algorithmus liefert hierbei die optimale Lösung, die von den approximativen Algorithmen angestrebt wird (siehe Abbildung 52b). Auffällig ist das sehr gute Abschneiden der FullAncestry-Strategie. Die Anzahl der gefundenen HHHs ist fast optimal. Die PartialAncestry Strategie dahingegen findet fast konstant 10 HHHs mehr. Dies bestätigt die Befürchtung aus dem Paper [13], dass PartialAncestry zwar prinzipiell weniger Vorgängerknoten und ihre Häufigkeit speichern muss, aber dadurch zusätzliche Informationen verliert und somit wiederum zu viele HHHs speichert. Das macht sich auch im Speicherverbrauch bemerkbar, da der PartialAncestry da auch schlechter abschneidet.

[2] Seitenaufrufe auf http://jwall.org



(a) Speicherverbrauch der HHH Strategien



(b) Anzahl ausgegebener HHHs

Abbildung 52: Evaluierung der FullAncestry-, PartialAncestry und NoStream-Strategien auf 250.000 IP-Adressen extrahiert aus einer jwall.org Log-Datei

6.2 Quantile

Bei Quantilen handelt es sich um Lagemaße bzw. Rangstatistiken von Beobachtungsreihen. Das wohl bekannteste Quantil ist der sogenannte Median. Ein Element wird Median genannt, wenn 50% aller Elemente einer Beobachtungsreihe kleiner als das Median-Element sind. Dieses Element wird auch als 0.5-Quantil bezeichnet. Weitere wichtige Quantile sind das 0.25- und das 0.75-Quantil. Sie werden als unteres bzw. oberes Quartil bezeichnet. Man spricht von einem ϕ-Quantil-Element, wenn $(\phi \cdot 100)\%$ aller Beobachtungen einen Wert kleiner oder gleich dem Wert dieses Elements haben.

Für die Berechnung eines solchen Elements werden die n Beobachtungen aufsteigend sortiert und mit Rängen versehen. Das absolute Minimum hat somit den Rang 1 und das Maximum den Wert n. Das ϕ-Quantil-Element ist dann genau das Element, dessen Rang ϕn beträgt. Dieses naive Verfahren ist allerdings nicht geeignet, um Rangstatistiken von Datenströmen zu verwalten. Es ist offensichtlich, dass für die Bestimmung exakter Quantile alle Elemente gespeichert werden müssen.
Deshalb wurden in diesem Projekt im Wesentlichen approximative Quantilalgorithmen implementiert. Zu Testzwecken ist das oben beschriebene exakte Verfahren allerdings unter dem Namen **Exact Quantiles** zu finden.

Die Klasse der approximativen Quantilalgorithmen besteht aus zwei unterschiedlichen Typen von Algorithmen:

- deterministische Quantilschätzer
- probabilistische Quantilschätzer

Ein deterministischer Quantilschätzer garantiert, dass jederzeit für beliebige Anfragen eines ϕ-Quantils eine Antwort geliefert wird. Diese Antwort ist nicht zwangsläufig identisch mit dem exakten Quantil. Die maximale Abweichung des geschätzten Quantils ist jedoch nach oben durch einen Fehlerparameter ϵ beschränkt. Dieser maximale Fehler bezieht sich nur auf die Abweichung der Ränge von Elementen, nicht jedoch auf die Differenz von geschätztem Wert zu wahrem Wert. Dies bedeutet, der geschätzte Rang r_{est} befindet sich in einer ϵ-Umgebung des wahren Rangs r_{true}. Es gilt Ungleichung 14. Abbildung 53 verdeutlicht dies anhand eines Beispiels mit $n = 10$ Elementen bei einem maximalen Fehler von $\epsilon = 0.2$ für das 0.7-Quantil.

$$r_{true} - \epsilon n \leq r_{est} \leq r_{true} + \epsilon n \tag{14}$$

Das exakte 0.7-Quantil-Element hat Rang 7. Bei 10 Beobachtungen und einem maximalen Fehler von $\epsilon = 0.2$ beträgt die maximale Abweichung zwei Ränge — gültige ϵ-approximative 0.7-Quantil-Elemente sind also Elemente mit Rängen zwischen 5 und 9.
Das Beispiel verdeutlicht außerdem, dass Quantile keinen Anspruch erheben „Wertstatistiken“ zu sein. Der absolute Fehler bezüglich der Werte ist nicht beschränkt. Probabilistische Quantilschätzer können, im Gegensatz zu deterministischen Verfah-

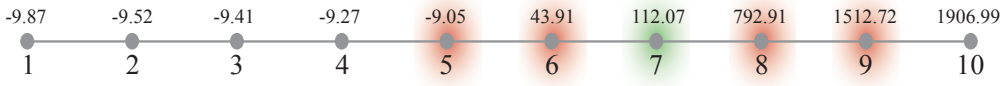


Abbildung 53: Gültige 0.7-Quantil-Elemente bei einem maximalen Fehler von $\epsilon = 0.2$

ren, nicht immer eine Antwort liefern. Sie verwenden randomisierte Datenstrukturen, deren Inhalte zufällig bestimmt oder abgefragt werden. In ungünstigen Fällen ist somit eine Aussage über bestimmte Ränge oder Rangabschnitte nicht möglich und der Algorithmus kann keine Schätzung abgeben. Die Frage ob eine Schätzung abgegeben werden kann, hängt oft auch von der Fehlerschranke ab. Genauere Ergebnisse sind aufwendiger, benötigen mehr Informationen. In der Regel kann bei probabilistischen Algorithmen die Versagenswahrscheinlichkeit δ eingestellt werden. Dabei gilt grundsätzlich, dass eine geringere Versagenswahrscheinlichkeit einen größeren Speicherbedarf mit sich bringt.

6.2.1 Simple Quantiles

Bei dem Simple Quantiles Algorithmus handelt es sich um ein einfaches Verfahren zum Schätzen von Quantilen. Das Verfahren ist im Wesentlichen unabhängig von der Eingabemenge, setzt allerdings voraus, dass die Daten nicht aus einer endlastigen Verteilung resultieren. So ist dieses Verfahren unter anderem nicht für pareto- oder logarithmisch normal-verteilte Datenströme geeignet.
Die Schätzung eines ϕ-Quantils bzw. des Rangs eines ϕ-Quantil-Elements erfolgt mit diesem Verfahren indirekt. Hierzu wird die statistische Erkenntnis verwendet, dass bei Stichproben unbekannter, nicht endlastiger Verteilungen mit gleichbleibender Kardinalität die Grenzen eines α-Konfidenzintervalls für ein ϕ-Quantil als konstant angenommen werden können (siehe Kapitel 4.5.1 aus [25] für detailliertere Informationen). Der Umfang der Stichprobe ist dabei von dem gewünschten α des Konfidenzintervalls abhängig.

Zur Bestimmung der Intervallgrenzen werden die $\mu_{1-\frac{\alpha}{2}}$-Quantile der Standardnormalverteilung herangezogen. Wenn diese passend zum gewählten α berechnet wurden, können die obere und untere Grenze für das α-Konfidenzintervall zum ϕ-Quantil mit Gleichung 15 und 16 berechnet werden. Hierbei ist n die Kardinalität der Stichprobe.

$$r_l := n\phi - \mu_{1-\frac{\alpha}{2}}\sqrt{n\phi(1-\phi)} \tag{15}$$

$$r_u := n\phi + \mu_{1-\frac{\alpha}{2}}\sqrt{n\phi(1-\phi)} \tag{16}$$

Abbildung 54: Bestimmung des 0.15-Quantils bei gewählter Konfidenz von $\alpha = 0.01$.

Um das eigentliche ϕ-Quantil-Element zu bestimmen, wird nun die Stichprobe sortiert, um anschließend das Element mit Rang $r_l + \lceil \frac{r_u - r_l}{2} \rceil$ auszuwählen und zurückzugegeben. In Abbildung 54 ist das α-Konfidenzintervall des ϕ-Quantils für $\alpha = 0.01$ und $\phi = 0.15$ dargestellt. Für ein α von 1% werden 910 Elemente in der Stichprobe gehalten. Es ist sowohl der Rang der unteren (108) und oberen Grenze (165) als auch des ausgewählten ϕ-Quantil-Elements (137) eingetragen.

Das Verfahren ist effizient bezüglich Rechenzeit und damit für den Einsatz in hochkapazitiven Datenströmen geeignet. Allerdings kann aufgrund der Datenunabhängigkeit nur in sehr speziellen Fällen der Fehler beschränkt werden. Es wird deshalb davon abgeraten das Verfahren zu verwenden, wenn sehr genaue Quantile benötigt werden.

Knotenklasse

edu.udo.cs.pg542.util.node.learner.Quantiles

Eingaben

Schlüssel	Typ	Bedeutung
event:learner:input	Double	Ein neues Element, dass in die Datenstruktur eingepflegt bzw. gelernt werden soll.

Ausgaben

Schlüssel	Typ	Bedeutung
event:learner:output	Selective-Description-Model	Das Model enthält das gelernte Quantil.

Parameter

Schlüssel	Typ	Opt.	Bedeutung
epsilon	Double	Nein	Der Parameter ϵ beschränkt den Fehler. Da das Verfahren ein Konfidenzintervall aufspannt in dem das Quantil vermutet wird, entspricht ϵ hier der Konfidenz. Der Speicheraufwand steigt exponentiell mit sinkendem Fehler.

6.2.2 Quantilalgorithmus nach Greenwald-Khanna

Mit *GKQuantiles* steht eine Implementierung des populären GK-Algorithmus nach Greenwald-Khanna [21] zur Verfügung. Der Algorithmus zeichnet sich durch seine große Speichereffizienz von $O\left(\frac{1}{\epsilon}\log \epsilon n\right)$ aus. Es ist somit zwar nicht möglich, potentiell unendlich lange Datenströme zu behandeln, allerdings ist der GK-Algorithmus der Ausgangspunkt von vielen Quantilalgorithmen für Datenströme (siehe z.B. Kapitel 6.2.3 und 6.2.4).

Das Verfahren verwaltet die Elemente des Datenstroms in sogenannten Tupeln. Ein Tupel t_i enthält neben dem Wert v_i eines Elements zwei weitere Größen g_i und Δ_i, die indirekte Ranginformationen enthalten. Hierbei ist g_i der Offset eines Tupels und berechnet sich anhand der Differenz des minimalen Rangs des Tupels t_i und seinem direkten Vorgänger t_{i-1}:

$$g_i := r_{min}\left(v_i\right) - r_{min}\left(v_{i-1}\right) \tag{17}$$

Die Differenz zwischen maximalem und minimalem Rang von t_i wird als Spanne bezeichnet und in Δ_i verwaltet. Hierfür wird folgende Formel verwendet:

$$\Delta_i := r_{max}\left(v_i\right) - r_{min}\left(v_i\right) \tag{18}$$

Das Einfügen von Elementen in die Datenstruktur des Algorithmus erfolgt, indem ein neues Tupel t_i angelegt wird, das den Wert v_i des Elements übernimmt. Dazu wird der Offset g_i auf 1 gesetzt. Anschließend wird in der bestehenden Datenstruktur das aufeinanderfolgende Tupelpaar t_j, t_k gesucht, so dass $v_j \leq v_i < v_k$ gilt. Der Wert für die Spanne von t_i wird auf $\Delta_i = g_k + \Delta_k - 1$ gesetzt. Ausnahmen zu dieser Regel sind Elemente die das aktuelle Minimum unter- oder das aktuelle Maximum überschreiten. Hier wird $\Delta = 0$ gesetzt, da diese Werte eindeutige Ränge haben müssen.

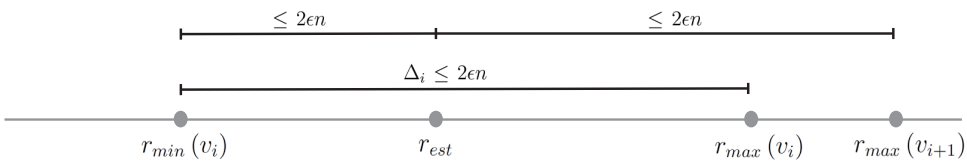


Abbildung 55: Veranschaulichung der Rang-Bedingung (Ungleichung 19)

Mit diesen indirekten Ranginformationen ist es möglich, absolute Werte für den minimalen und maximalen Rang eines Tupels zu bestimmen:

$$\begin{aligned} r_{min}(v_i) &= \sum_{j \leq i} g_j \\ &= r_{min}(0) - 0 + r_{min}(1) - r_{min}(0) + \cdots + r_{min}(v_i) - r_{min}(v_{i-1}) \end{aligned}$$

$$\begin{aligned} r_{max}(v_i) &= \left(\sum_{j \leq i} g_j \right) + \Delta_i \\ &= r_{min}(v_i) + \Delta_i \\ &= r_{min}(v_i) + r_{max}(v_i) - r_{min}(v_i) \end{aligned}$$

Die große Speichereffizienz erreicht der GK-Algorithmus dadurch, dass ein Tupel t_i i.d.R. mehrere Elemente des Datenstroms repräsentiert. Hierbei wird die maximale Kapazität eines Tupels durch die Überlegung vorgegeben, dass jedes ϕ-Quantil mit einem maximalen Fehler ϵ bestimmt werden kann, wenn ein Tupel höchstens $2\epsilon n$ Elemente enthält. Der maximale Fehler beträgt $r_{error} = \epsilon n$. Wenn also ein ϕ-Quantil gesucht ist, kann jeder Rang zurückgegeben werden für den die Ungleichung 19 gilt. Gilt $\Delta_i \leq 2\epsilon n$ für alle i, dann kann jederzeit ein gültiges r_{est} gefunden werden. Es muss lediglich ein Rang r bestimmt werden für den gilt: $r_{min}(v_i) \leq r < r_{max}(v_{i+1})$. Dieser Rang ist ein gültiges r_{est} und ist nicht notwendigerweise eindeutig (Abbildung 55). Für einen ausführlichen Beweis wird auf die Arbeit von Greenwald und Khanna [21] verwiesen.

$$r_{true} - r_{error} \leq r_{est} \leq r_{true} + r_{error} \tag{19}$$

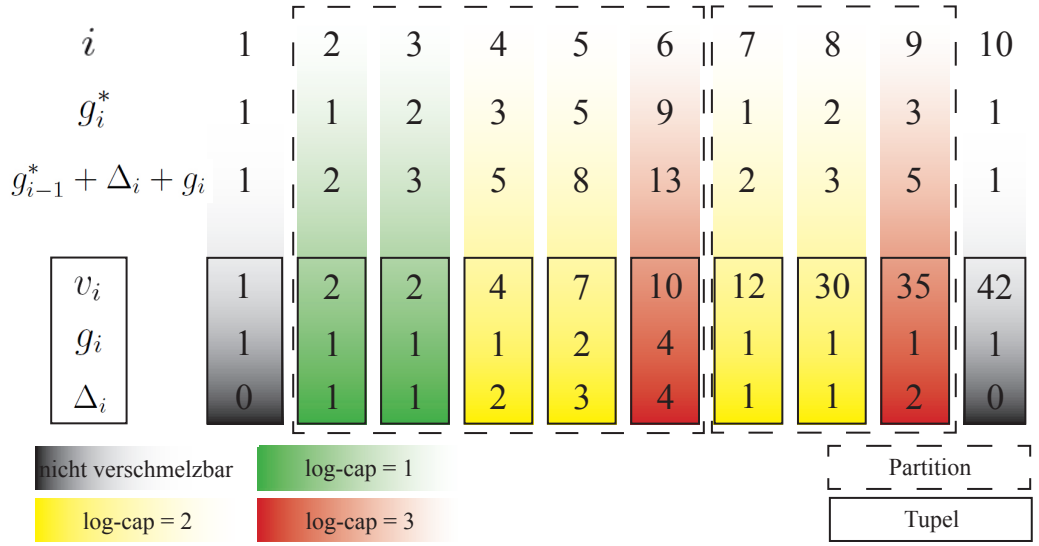


Abbildung 56: Partitionierung der Datenstruktur

Um überflüssige Tupel aus der Datenstruktur zu entfernen wird nach jeweils $\lceil \frac{1}{2\epsilon} \rceil$ Einfüge-Operationen die Datenstruktur vom Algorithmus durchwandert, um Tupel ineinander zu verschmelzen.
Ein aufeinander folgendes Paar von Tupeln (t_{i-1}, t_i) muss eine Reihe von Bedingungen erfüllen, damit ein Verschmelzen den Fehler ϵ nicht verletzt. Zunächst werden die Tupel in Partitionen aufgeteilt. Innerhalb einer Partition sind sowohl die Werte als auch die Kapazitäten der Tupel monoton steigend. Die Kapazität eines Tupels ist hierbei die Anzahl der Elemente die ein Tupel aufnehmen kann bevor es voll ist. In Ungleichung 19 wurde bereits verdeutlicht, dass die Spanne eines Tupels höchstens $2\epsilon n$ betragen darf. Da Δ_i fest ist, folgt daraus, dass die Kapazität eines Tupels $2\epsilon n - \Delta_i$ beträgt.
Wenn für ein Tupelpaar (t_i, t_{i+1}) gilt, dass $cap_i > cap_{i+1}$ ist, ist eine Partitionsgrenze erreicht. Von der Partitionierung ausgenommen sind das Minimum und das Maximum. Abbildung 56 zeigt eine solche Partitionierung.
Die Partitionen werden anschließend aufsteigend durchlaufen. Für jedes Tupelpaar (t_{i-1}, t_i) werden die drei Bedingungen 20, 21 und 22 geprüft. Hierbei wird die Größe g_i^* benötigt. In dieser Variable wird die Summe aller g_j mit $j < i$ innerhalb einer Partition verwaltet.

$$v_{i-1} < v_i \tag{20}$$

$$\log(g_{i-1} + \Delta_{i-1}) \leq \log(g_i + \Delta_i) \tag{21}$$

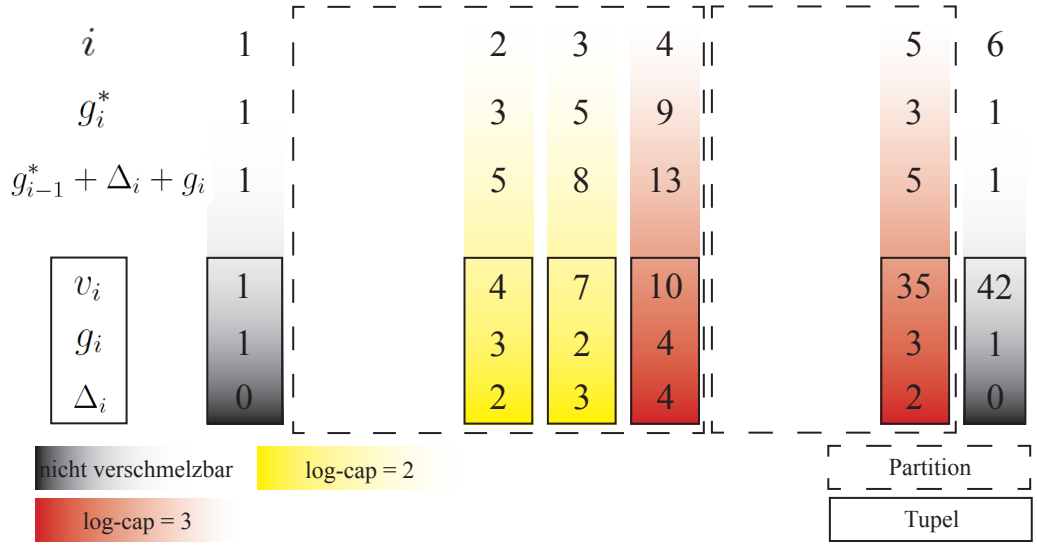


Abbildung 57: Die Datenstruktur aus dem Beispiel nach dem Verschmelzen

$$g_{i-1}^* + \Delta_i + g_i < 2\epsilon n \tag{22}$$

Damit keinerlei Ranginformationen durch das Entfernen von Tupeln t_{i-1} verloren gehen, wird $t_i = (v_i, g_i, \Delta_i)$ ersetzt durch $\dot{t}_i = \left(v_i, g_{i-1}^* + g_i, \Delta_i\right)$, da sich durch das Absorbieren weiterer Tupel mit kleinerem Wert nur der minimale Rang von t_i ändert. Der Offset steht also für die maximale Anzahl an Tupeln, die zwischen zwei aufeinanderfolgenden Tupeln gelöscht wurden.
Für das Beispiel aus Abbildung 56 wird angenommen, dass $2\epsilon n = 6$ gilt. Nach Anwendung der Regeln 20, 21 und 22 ergibt sich nach dem Verschmelzen die Abbildung 57.

Trotz der großen Speichereffizienz ist dieser Algorithmus nur bedingt onlinefähig. Dies liegt daran, dass der Speicherbedarf weiterhin - wenn auch nur logarithmisch - von der Eingabelänge abhängt. Der GK-Algorithmus eignet sich für Datenströme mit endlicher, nicht notwendigerweise a priori bekannter Anzahl von Elementen. Zu jeder Zeit werden höchstens $\left(\frac{11}{2\epsilon} \log\left(2\epsilon n\right)\right)$ Tupel in der Datenstruktur gehalten [21].
Für den Einsatz auf potentiell unendlich langen Datenströmen sollte allerdings auf Verfahren zurückgegriffen werden, die den GK-Algorithmus als Basis für Quantilschätzer mit konstanter oberer Grenze für den Speicherbedarf verwenden. Hierzu zählen unter anderem *Window Sketch Quantiles* (Kapitel 6.2.3) und *Ensemble Quantiles* (Kapitel 6.2.4).

Knotenklasse

edu.udo.cs.pg542.util.node.learner.Quantiles

Eingaben

Schlüssel	Typ	Bedeutung
event:learner:input	Double	Ein neues Element, dass in die Datenstruktur eingepflegt bzw. gelernt werden soll.

Ausgaben

Schlüssel	Typ	Bedeutung
event:learner:output	Selective-Description-Model	Das Model enthält das gelernte Quantil.

Parameter

Schlüssel	Typ	Opt.	Bedeutung
epsilon	Double	Nein	Der Parameter ϵ steht für den maximalen Fehler. Dieser Fehler bezieht sich auf die Anzahl der Ränge die das geschätzte Element vom wahren Element abweichen darf. Mit sinkendem Fehler nimmt der Rechenaufwand zu.

6.2.3 Window Sketch Quantiles

Das Verfahren Window Sketch Quantiles ist eine Erweiterung des Quantilalgorithmus von Greenwald und Khanna (Kapitel 6.2.2). Die Implementierung basiert auf einer Arbeit von Arasu und Manku [2] und realisiert einen für Datenströme geeigneten Quantilschätzer. Dies wird erreicht indem der Speicherbedarf konstant gehalten wird. Hierzu wird ein Fenster aufgespannt durch das der Datenstrom geleitet wird. Das resultierende Modell kann verwendet werden, um jederzeit Rangstatistiken für

Abbildung 58: Ein Vektor für beliebige ϕ-Quantile (rot) mit Fehler $\epsilon = 0.125$

den sich im Fenster befindlichen Teil des Datenstroms zu erstellen. Die Breite des Fensters ist konfigurierbar.
Der Algorithmus von Arasu und Manku [2] sieht vor, dass vom gesamten sich im Fenster befindlichen Datenstrom Kopien angefertigt werden. Die Anzahl der Kopien wird durch den gewünschten maximalen Fehler ϵ bestimmt und beträgt $L = \lceil \log_2 \left(\frac{4}{\epsilon}\right) \rceil$. Auf jeder Ebene wird eine Zusammenfassung des Datenstroms mit unterschiedlichem Fehler ϵ_l verwaltet. Die Genauigkeit ist dabei aufsteigend, d.h. der maximale Fehler auf einer beliebigen Ebene ist immer größer, als der auf der nächsthöheren Ebene. Die Kopien werden auf jeder Ebene in Blöcke aufgeteilt. Die Anzahl der Elemente pro Block ist auf einer Ebene fest, jedoch ist diese Anzahl abhängig von ϵ_l und nimmt mit steigenden Ebenen zu.

Solange ein Block nicht vollständig gefüllt ist, verwaltet eine Instanz des GK-Algorithmus die Elemente des Datenstroms. Wenn ein Block seine maximale Anzahl an Elementen erreicht hat, wird eine Zusammenfassung dieses Blocks erstellt. Dazu wird ein Vektor mit ϕ-Quantilen mit äquidistanten Abständen erzeugt. Weil die maximale Genauigkeit auf jeder Ebene l genau ϵ_l beträgt und der Fehler der Zusammenfassung höchstens ϵ_l betragen darf, werden die ϕ-Quantile an den Stellen $[\epsilon, 2\epsilon, 3\epsilon, ..., \alpha\epsilon]$ (mit $(1 - \epsilon) < \alpha\epsilon \leq 1$) geschätzt und bilden den Vektor (siehe Abbildung 58). Anschließend wird der Vektor in das Fenster eingefügt und eine neue Instanz des GK-Algorithmus für den neuen Block erzeugt. Abbildung 59 zeigt die Partitionierung der Kopien des Datenstroms im Fenster.
Die Blöcke haben den Status aktiv oder inaktiv. Solange sich alle Elemente eines Blocks innerhalb des Fensters befinden ist ein Block aktiv. Sobald das erste Element eines Blocks das Fenster verlässt ist dieser Block veraltet. Er wird inaktiv und aus dem Fenster entfernt. Inaktiv sind auch Blöcke in der Entstehung, d.h. für die noch nicht alle Elemente das Fenster erreicht haben.

Die entstehende Datenstruktur kann nun zur Schätzung von Quantilen verwendet werden. Dazu wird eine disjunkte Menge von aktiven Blöcken ausgewählt, die den maximal möglichen Teil des Fensters abdeckt. Außerdem soll die ausgewählte Menge den Gesamtfehler ϵ minimieren. Das wird erreicht, indem möglichst große Blöcke

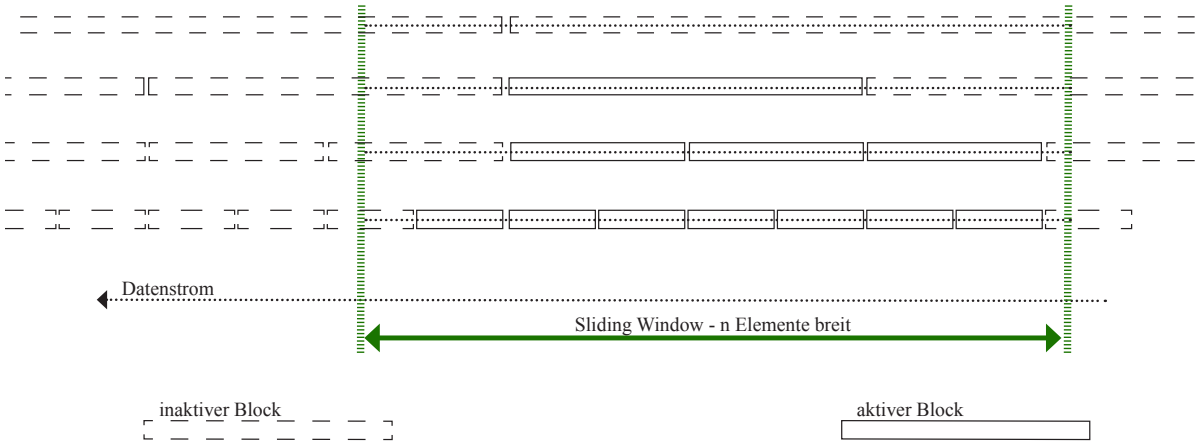


Abbildung 59: aktive und inaktive Blöcke auf unterschiedlichen Ebenen

ausgewählt werden. In der Praxis beginnt der Algorithmus mit dem größten aktiven Block und deckt die übrigen Teile des Datenstroms mit Blöcken aus unteren Ebenen ab. Abbildung 60 verdeutlicht das Verfahren und zeigt die Auswahl des Fensters mit minimalem Fehler ϵ rot markiert.

Wenn eine solche Zusammenfassung des Fensters gefunden ist, müssen die Elemente der einzelnen Blöcke geeignet gewichtet werden. Durch die Gewichtung soll erreicht werden, dass ein großer Block mit vielen Elementen und kleinem ϵ_l bei der Schätzung eines ϕ-Quantils mehr Bedeutung hat als kurze Blöcke mit hohem ϵ_l aus niedrigeren Blöcken. Bei der praktischen Durchführung der Gewichtung weicht die Implementierung von Window Sketch Quantiles von der Arbeit von Arasu und Manku [2] ab.
Um einen Gesamtvektor zum Schätzen von ϕ-Quantilen zu erzeugen werden die Elemente der ausgewählten Blöcke sortiert in eine Liste eingefügt. Um Blöcke aus höheren Ebenen stärker zu gewichten werden die Elemente jedoch unterschiedlich oft eingefügt. Die Häufigkeit des Einfügens wird durch die Ebene des Blocks bestimmt. Elemente eines Blocks aus der untersten Ebene 0 werden einmal, Elemente aus einem Ebene-l-Block (l+1) mal eingefügt.
Zum Beantworten der Quantil-Anfrage genügt es nun den Wert des Elements an der Stelle ϕm auszugeben, wobei m die Kardinalität der Elemente in der oben beschriebenen Liste ist.

Im Testbetrieb hat sich herausgestellt, dass der Window Sketch Quantiles Ansatz nur bedingt für Datenströme mit hohem Durchsatz geeignet ist. Der Grund hierfür ist die entstehende Rechenlast. Es werden multiple Instanzen des GK-Algorithmus zeitgleich verwendet, da auf jeder Ebene jederzeit ein Block unvollständig ist.

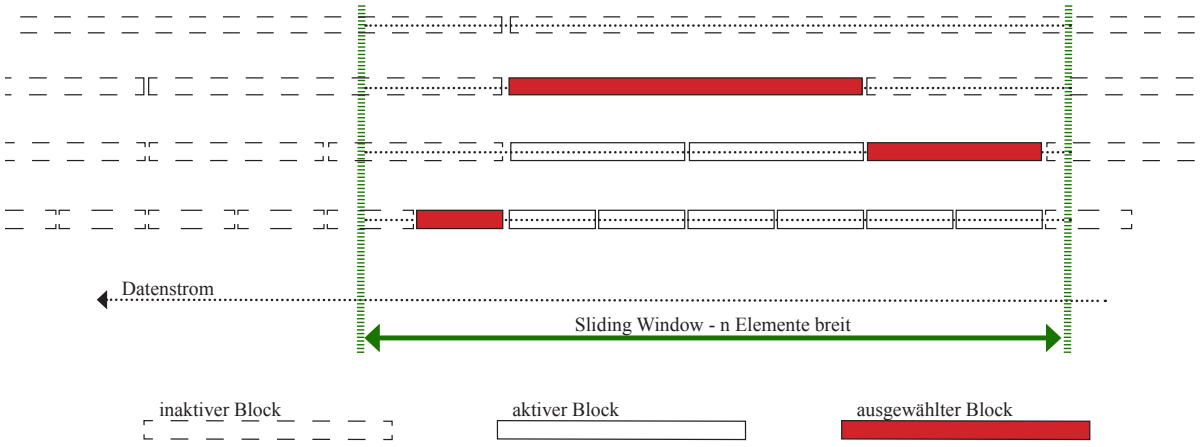


Abbildung 60: Menge von Blöcken mit minimalem Fehler ϵ

Des Weiteren nimmt die Rechenkapazität pro GK-Instanz mit steigender Ebene zu. Dies ergibt sich aus dem sinkenden Fehlerparameter ϵ_l. Da die Zusammenfassung dem vorgegebenen Fehler ϵ genügen muss und dieser aus den Einzelfehlern ϵ_l der unterschiedlichen Ebenen zusammengesetzt wird, unterschreiten höhere Ebenen sogar den gewählten Gesamtfehler ϵ und verursachen rechenintensive GK-Instanzen.
Ein weiterer Faktor der den Rechenaufwand erhöht ist die Verwendung eines *sliding windows*, weil nach jedem neuen Element im Datenstrom dieses Fenster aktualisiert werden muss. Dazu müssen auf sämtlichen Ebenen die ältesten Blöcke daraufhin überprüft werden, ob ein Element das Fenster bereits verlassen hat und somit der ganze Block verworfen werden muss.
Neben der hohen Rechenkapazität des Algorithmus wirkt es sich in vielen Anwendungen nachteilig aus, dass Rangstatistiken nur über einen Teil des Datenstroms geführt werden. Gerade in hochkapazitiven Datenströmen wird dieser Ausschnitt schnell zu einem unbedeutenden Bruchteil des Gesamtstroms. Auf einen *concept drift* der Daten wird so zwar schnell durch ein angepasstes Modell reagiert, es besteht jedoch die Gefahr, dass eine temporäre Häufung von Ausreißern die Güte der geschätzten ϕ-Quantile stark verschlechtert.

Um diese Nachteile zu umgehen bzw. abzuschwächen wurde ein weiteres Verfahren implementiert, das auf dem GK-Algorithmus basiert. Dieser *Ensemble Quantiles* genannte Ansatz ist im nachfolgenden Kapitel 6.2.4 dokumentiert.

Knotenklasse

edu.udo.cs.pg542.util.node.learner.Quantiles

Eingaben

Schlüssel	Typ	Bedeutung
event:learner:input	Double	Ein neues Element, dass in die Datenstruktur eingepflegt bzw. gelernt werden soll.

Ausgaben

Schlüssel	Typ	Bedeutung
event:learner:output	Selective-Description-Model	Das Model enthält das gelernte Quantil.

Parameter

Schlüssel	Typ	Opt.	Bedeutung
epsilon	Double	Nein	Der Parameter ϵ steht für den maximalen Fehler. Dieser Fehler bezieht sich auf die Anzahl der Ränge die das geschätzte Element vom wahren Element des aktuellen Fensters abweichen darf. Mit sinkendem Fehler nimmt der Rechen- und Speicheraufwand zu.
windowSize	Integer	Ja	Dieser Wert beziffert die Anzahl der Elemente die das Fenster maximal fasst. Falls keine Zweierpotenz angegeben wird, wird der Wert aus Gründen der Performance auf die nächst höhere Zweierpotenz aufgerundet.

6.2.4 Ensemble Quantiles

Vorbemerkung Das im Folgenden beschriebene Verfahren der Ensemble Quantiles ist nach bestem Wissen der Projektgruppe bislang nicht dokumentiert. Während

Abbildung 61: Ein Ensemble aus Teilmodellen

der Implementierungsphase wurde ein Quantilalgorithmus gesucht, der sowohl konstanten Speicherbedarf hat als auch hochkapazitive Datenströme verarbeiten kann. Nachdem mit dem Window Sketch Quantile Algorithmus ein mögliches Verfahren implementiert wurde, stellte sich heraus, dass der Rechenbedarf bei diesem Verfahren relativ hoch ist. Der Window Sketch Quantiles Algorithmus (Kapitel 6.2.3) beschränkt den Speicherbedarf des Greenwald-Khanna Algorithmus (Kapitel 6.2.2) indem der Datenstrom in Blöcke mit endlicher Länge aufgeteilt und jeder Block nach einer gewissen Zeit verworfen wird um Speicher frei zu räumen. Der Ensemble Quantiles Algorithmus begrenzt den Speicher ebenfalls durch eine Einteilung in feste Blöcke. Allerdings sind unterschiedliche Update-Strategien verfügbar. Um den Rechenbedarf zu verringern wurde darauf verzichtet Kopien des Datenstroms anzulegen. Bei gleichem Speicherbedarf deckt ein Modell des Ensemble Quantile Algorithmus also einen größeren Teil des Datenstroms ab, als ein Modell des Window Sketch Quantil Algorithmus.

Umsetzung Der Algorithmus teilt den Datenstrom in Blöcke auf. Jeder Block enthält die identische, vom Benutzer vorgegebene Anzahl aufeinander folgender Elemente des Datenstroms. Die Partitionierung des Datenstroms erfolgt disjunkt. Ähnlich zum Window Sketch Quantiles Verfahren werden alle Elemente eines nicht vollständig gefüllten Blocks von einer Instanz des GK-Algorithmus verwaltet.
Sobald ein Block seine maximale Anzahl an Elementen erreicht hat, wird eine Zusammenfassung in Form eines Vektors analog zum Window Sketch Quantiles Ansatz erstellt. Jede dieser Zusammenfassungen bildet ein Teilmodell des Datenstroms.
Um die maximale Genauigkeit des GK-Algorithmus auszunutzen und gleichzeitig den maximalen Fehler nicht zu überschreiten genügt es die Anfragen mit Abstand des Fehlermaßes ϵ zu verwenden. Der Vektor (Abbildung 58) enthält somit die ϕ-Quantile an den Stellen $[\epsilon, 2\epsilon, 3\epsilon, ..., \alpha\epsilon]$ (mit $(1 - \epsilon) < \alpha\epsilon \leq 1$).

Eine Menge von n Vektoren bildet das Ensemble. Durch die Begrenzung der maximalen Anzahl von Teilmodellen wird der Speicherbedarf des Algorithmus konstant gehalten. Solange das Modell aus weniger als n Vektoren besteht, werden neue Vektoren dem Ensemble hinzugefügt. Sobald der $(n + 1)$. Block vollständig gefüllt ist, wird eine Update-Methode aufgerufen. Für das Aktualisieren des Ensembles stehen sieben unterschiedliche Methoden zur Verfügung. Der Benutzer kann mit der Wahl der Update-Methode Einfluss auf die Güte der geschätzten Quantile nehmen und Apriori Wissen über die Verteilungsfunktion des Datenstroms einfließen lassen. Je nach Wahl

der Update-Methode besteht das Ensemble aus unterschiedlichen Teilmodellen des Datenstroms. Ein typisches Gesamtmodell ist in Abbildung 61 dargestellt.
Im Folgenden werden die einzelnen Update-Methoden kurz vorgestellt. Wenn der Abstand zwischen Teilmodellen Kriterium ist, wird der quadratische Abstand von Vektoren verwendet:

$$\sqrt{\sum_{\forall i} (x_i - y_i)^2} \tag{23}$$

Ersetzen des Blocks mit der geringsten Ähnlichkeit zum aktuellen Block Diese Update-Methode ersetzt den Vektor mit dem größten Abstand zum neuen Vektor. Von Vorteil ist die Auswahl dieser Update-Methode wenn ein *concept drift* in den Daten möglichst schnell vom Modell erfasst werden soll, was dadurch erreicht wird, dass der aktuellste Teil des Datenstroms als Referenz verwendet wird.

Ersetzen des ältesten Blocks im Ensemble Durch das Verwenden dieser Update-Methode entsteht im Wesentlichen ein Modell wie beim Window Sketch Quantile Algorithmus. Auch hier wird ein concept drift schnell vom Modell adaptiert, allerdings ist das Modell nicht so robust gegen Ausreißer wie bei der Ersetzung des Blocks mit der geringsten Ähnlichkeit.

Ersetzen des Blocks mit der geringsten Ähnlichkeit zu einer Stichprobe des Datenstroms Das Verfahren ist analog zum Ersetzen des Blocks mit der geringsten Ähnlichkeit. Lediglich die Referenz ändert sich. Es wird nicht der aktuellste Block zur Bildung des Abstands verwendet sondern eine Stichprobe des Datenstroms. Diese Stichprobe wird probabilistisch aus den Elementen des Datenstroms gezogen. Für jedes Element wird mit einer Wahrscheinlichkeit in Abhängigkeit von einem konfigurierbaren *sampling parameter* entschieden, ob es der Stichprobe hinzugefügt wird oder nicht. Die Anzahl der Elemente einer Stichprobe ist ebenfalls durch die maximale Anzahl von Elementen eines Blocks beschränkt. Je nach Wahl des sampling parameters enthält die Stichprobe also Elemente die zur Bildung von unterschiedlichen Teilmodellen verwendet wurden. Diese Update-Methode kann von Vorteil sein, wenn ein periodischer Drift des Datenstroms vorliegt.

Ersetzen eines zufälligen Blocks Diese Update-Methode verwirft ein zufällig ausgewähltes Teilmodell zugunsten des neuen Vektors.

Verschmelzen der beiden ältesten Blöcke Wenn diese Update-Methode ausgewählt wurde, werden bei jedem Update die Informationen des ältesten Blocks in den zweitältesten Block eingefügt. Das Verschmelzen von Blöcken geschieht durch die

Mittelwertbildung der jeweiligen Vektoren. Mit dieser Update-Methode wird eine Alterungsfunktion eingefügt, da ältere Blöcke nie vollkommen aus dem Modell entfernt werden, ihr Einfluss allerdings stetig abnimmt.

Verschmelzen der beiden ähnlichsten Blöcke Das Ensemble wird nach den beiden Vektoren mit geringstem Abstand zueinander durchsucht. Auch hier werden die Vektoren verschmolzen indem dimensionsweise der Mittelwert der Vektoren gebildet wird. Wenn ein Datenstrom periodischen Wechseln unterliegt, kann der Informationsverlust durch diese Update-Methode minimiert werden. Es besteht allerdings die Gefahr, dass sehr häufige Elemente in ihrer Gewichtung abgeschwächt werden, da diese zu ähnlichen Vektoren führen.

Verschmelzen der beiden unähnlichsten Blöcke Im Gegensatz zum Verschmelzen der ähnlichsten Blöcke werden bei der Verwendung dieser Update-Methode die beiden Vektoren mit dem größten Abstand zueinander verschmolzen. Das Modell wird dadurch robuster gegen Ausreißer und sehr häufige Elemente verlieren nicht ihre Gewichtung im Ensemble sondern werden vielmehr höher gewichtet.

Die Bestimmung von ϕ-Quantilen ist denkbar einfach. Alle im Ensemble enthaltenen Vektoren werden konkateniert und sortiert. Im Gegensatz zum Ansatz Window Sketch Quantiles wird auch der aktuelle, nicht vollständig gefüllte Block bei der Schätzung des Quantils mit einbezogen. Hierzu wird dem Ensemble temporär eine Zusammenfassung des aktuellen Blocks hinzugefügt. In Anlehnung an das exakte Verfahren zur Bestimmung von Quantilen wird nun der Wert des Elements an Rang ϕm (mit $m \hat{=}$ Anzahl aller Elemente in der konkatenierten Zusammenfassung) als ϕ-Quantil zurückgegeben.

Ein wesentlicher Vorteil von Ensemble Quantiles gegenüber Window Sketch Quantiles ist die höhere Performance. Dies gilt - identische Fensterbreite vorausgesetzt - sowohl für die Speicher- als auch für die Rechenkapazität. Während die Anzahl der Instanzen von GK-Algorithmen bei Window Sketch Quantiles $L = \lceil \log_2 \left(\frac{4}{\epsilon}\right) \rceil$ beträgt und somit mit sinkendem Fehler zunimmt, benötigt Ensemble Quantiles konstant eine Instanz des GK-Algorithmus für den aktuellen Block.
Außerdem entfällt das regelmäßige Aktualisieren des Fensters nach jedem neuen Element. Ein Aktualisieren ist bei Ensemble Quantiles lediglich dann notwendig, wenn der aktuelle Block vollständig gefüllt wurde.
Des Weiteren ist die für eine Quantilanfrage verwendete Zusammenfassung jederzeit implizit durch das Ensemble gegeben und muss nicht erst durch Auswahl geeigneter Blöcke konstruiert werden.
Es ist zu erwarten, dass die Güte der geschätzten Quantile bei Verwendung von Ensemble Quantiles gegenüber dem Ansatz in Window Sketch Quantiles zunimmt, da zeitlich weiter zurückliegende Informationen mit einbezogen werden können. Eine

maximale Fehlerschranke kann - wie auch bei Window Sketch Quantiles - allerdings nur solange garantiert werden, wie keine Blöcke ersetzt oder verschmolzen wurden.

Knotenklasse

edu.udo.cs.pg542.util.node.learner.Quantiles

Eingaben

Schlüssel	Typ	Bedeutung
event:learner:input	Double	Ein neues Element, dass in die Datenstruktur eingepflegt bzw. gelernt werden soll.

Ausgaben

Schlüssel	Typ	Bedeutung
event:learner:output	Selective-Description-Model	Das Model enthält das gelernte Quantil.

Parameter

Schlüssel	Typ	Opt.	Bedeutung
epsilon	Double	Nein	Der Parameter ϵ steht für den maximalen Fehler. Dieser Fehler bezieht sich auf die Anzahl der Ränge die das geschätzte Element vom wahren Element abweicht, ist bei diesem Verfahren jedoch eher ein Richtwert.
chunkSize	Integer	Ja	Dieser Wert beziffert die Anzahl der Elemente die ein Block fasst. Wenn ein Block diese Anzahl von Elementen enthält, wird eine Zusammenfassung erstellt, die in das Ensemble eingepflegt wird.

ensembleSize	Integer	Ja	Mit der Angabe dieses Wertes wird die Anzahl der Teilmodelle, aus der sich das Gesamtmodell zusammensetzt, festgelegt. Der Speicherbedarf des Algorithmus steigt linear mit diesem Parameter.
sampleRatio	Double	Ja	Eine Angabe der *Sample-Ratio* ist nur sinnvoll wenn die gewählte Update-Methode dem Ersetzen des Blocks mit der geringsten Ähnlichkeit zu einer Stichprobe des Datenstroms entspricht. Es wird in diesem Fall eine Stichprobe erstellt, die als Referenzblock für das Aktualisieren verwendet wird. Eine *Sample-Ratio* von z.B. 0.5 bedeutet, dass im Schnitt jedes zweite Element Teil der Stichprobe wird.
ensembleUpdateMode	String	Ja	Das Verfahren Ensemble Quantiles ermöglicht es dem Anwender *apriori Wissen* über die Verteilung der Daten in die Modellbildung einfließen zu lassen. Dies geschieht durch die Auswahl einer von sieben möglichen Strategien zur Aktualisierung des Ensembles. Alle hier verwendbaren Werte sind als statische Strings in der Klasse *EnsembleQuantiles* hinterlegt. Für genauere Informationen zur Funktionsweise der einzelnen Methoden sei auf die entsprechenden Absätze in diesem Kapitel verwiesen.

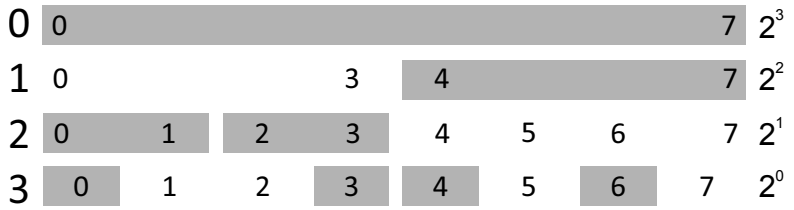


Abbildung 62: Aufbau eines SubsetTrees

6.2.5 Random Subset Sums

Das Verfahren Bei Random Subset Sum (RSS) handelt es sich um ein Verfahren zur Bestimmung von Quantilen. Der Algorithmus arbeitet probabilistisch und garantiert eine Fehlerwahrscheinlichkeit von ϵ und eine Versagenswahrscheinlichkeit von δ.

Im Allgemeinen werden Quantile direkt über ihren Rang bestimmt. Eine weitere Möglichkeit um ein Quantil zu bestimmen besteht darin, eine Hilfs-Datenstruktur zu erstellen und für jedes Element aus der Menge der möglichen Eingaben - dem Universum U - zu protokollieren, wie häufig es bereits vorgekommen ist. Bildet man nun in aufsteigender Reihenfolge sukzessive die Summe über diese Einträge, so ist das gewünschte p-Quantil das Element, mit welchem die Summe den Wert pN erreicht.

RSS knüpft an diese Idee an, arbeitet jedoch nicht mit Einzelwerten, sondern mit Hierarchien von dyadischen Intervallen von Werten, wie es in Abbildung 62 beispielhaft dargestellt ist. Auf der untersten Ebene der Hierarchie haben die Intervalle eine Breite von 2^0 und für jede nächsthöhere Ebene ist der Exponent um eins inkrementiert. Bei der Erzeugung der Hierarchie wird jedes Intervall mit einer Wahrscheinlichkeit von $\frac{1}{2}$ in die Datenstruktur aufgenommen. Im Folgenden wird eine solche Hierarchieebene *Subset*, die vollständige Hierarchie dieser Subsets *SubsetTree* genannt.

Bei einer Einfügeoperation wird je Subset des SubsetTrees geprüft, ob ein das Element enthaltendes Intervall im Subset vorhanden ist. Ist dies der Fall, wird der Zählwert dieses Intervalls um eins erhöht. Wird beispielsweise in den SubsetTree in Abbildung 62 der Wert 4 eingefügt, so werden die Zählwerte der Intervalle $[0, 7], [4, 7]$ und $[4]$ je um Eins inkrementiert.
Um eine Anfrage nach dem p-Quantil zu bedienen, wird im Gegensatz zum eingangs vorgestellten Verfahren nicht die Summe der Quantitäten der Einzelwerte gebildet, was über das unterste Subset im SubsetTree prinzipiell möglich wäre, sondern es wird mit der Kombination von Intervallen aus höchstmöglichen Subsets im SubsetTree gearbeitet. Dies stellt eine Nutzung möglichst großschrittiger Intervalle sicher. Angenommen, in Abbildung 62 sei die 4 das Element, welches pN erfüllt. Dann wird zunächst in der höchsten Hierarchieebene geprüft, ob der Zählwert des Intervalls $[0 - 7]$ bereits pN übersteigt. Ist dies der Fall, muss dieses Intervall für die Betrachtung verworfen werden, da das pN erfüllende Intervall einelementig sein muss. Aus

demselben Grund kann $[4-7]$ nicht gewählt werden. Die Summe der Zählwerte von $[0-1]$ und $[2-3]$ überschreitet pN noch nicht, daher können sie als höchstmögliche Intervalle der Hierarchie für die Bildung der Summe herangezogen werden. Da im selben Subset keine weiteren Intervalle vorhanden sind, wird auf die Intervalle der Breite 2^0 zurückgegriffen. Hier erreicht die 4 schließlich in Summe mit $[0-1]$ und $[2-3]$ den gewünschten Zählwert pN. Somit kann 4 als Quantil-Element ausgegeben werden.
Um eine Garantie für Genauigkeit und Versagenswahrscheinlichkeit der Ergebnisanfragen zu geben, wird nicht mit einem einzigen SubsetTree gearbeitet, sondern es werden $24 \log(\log(|U|)\,\delta) \log(|U|)\,\epsilon^2$ SubsetTrees erzeugt. Jeder SubsetTree resultiert in einer Schätzung, welche wiederum in Pakete von $8 \log(|U|)\,/\epsilon^2$ Elementen zusammengefasst werden. Die endgültige Schätzung ergibt sich schließlich als Median der Mittelwerte dieser Pakete. Der Erwartungswert der Summe der Elemente in einem Subset ist die Hälfte der Anzahl der im gesamten SubsetTree vorhandenen Elemente. Demnach wird die Anzahl der Elemente innerhalb eines Intervalls, das im Subset gewählt sein muss, als $2 \cdot |Subset| - |Tree|$ geschätzt.

Implementierung und Stream Adaption Die im Rahmen der Projektgruppe durchgeführte Implementierung umfasst fünf Klassen, mit Namen *Interval*, *Bucket*, *Subset*, *SubsetTree* und die Hauptklasse *RandomSubsetSums*.

Ein Bucket vereint $24 \log(\log(|U|)\,\delta) \log(|U|)\,\epsilon^2$ SubsetTrees. Subsets bilden eine zufällig gewählte Teilmenge von *Interval*-Objekten einer fest gewählten dyadischen Größe. SubsetTrees vereinen schließlich Subsets der Größen 2^0 bis $log(|U|)$.

Um die im Subset vorhandenen Intervalle zu repräsentieren wurden in einer ersten Implementierung Interval-Objekte in einer Liste benutzt. Bei der großen Anzahl der zu erstellenden Kopien bzw. SubsetTree-Instanzen kam es sehr schnell zu *out of memory* Exceptions. In der aktuellen Implementierung werden *BitSets* verwendet, um zu indizieren, ob ein Intervall sich im SubsetTree befindet. Dies wirkt sich auch positiv auf den Durchsatz aus.

Um dieses Verfahren auf Streams anzuwenden, ist es in eine $SlidingWindow$ Umgebung eingebettet worden. Dazu werden Buckets mit einer beliebigen aber fest gewählten Kapazität C_{Bucket} verwaltet. In den Bucket können also C_{Bucket} Elemente eingefügt werden, bis er als voll gilt. Ist ein Bucket voll, wird ein neuer Bucket erstellt und in das Sliding Window eingefügt, welches wiederum eine Kapazität $C_{SlidingWindow}$ besitzt. Ist diese überschritten, wird der älteste Bucket gelöscht. Anfragen über die Anzahl der Elemente in einem Intervall werden an jeden Bucket gestellt und aufsummiert. Diese Summe bildet die Antwort für die letzten $C_{Bucket} \cdot C_{SlidingWindow}$ Elemente des Streams.

Knotenklasse

edu.udo.cs.pg542.util.node.learner.Quantiles

Eingaben

Schlüssel	Typ	Bedeutung
event:learner:input	Double	Ein neues Element, dass in die Datenstruktur eingepflegt bzw. gelernt werden soll.

Ausgaben

Schlüssel	Typ	Bedeutung
event:learner:output	Selective-Description-Model	Das Model enthält das gelernte Quantil.

Parameter

Schlüssel	Typ	Opt.	Bedeutung
epsilon	Double	Nein	Das Epsilon bestimmt die Genauigkeit des Algorithmus. Man spannt damit einen „Schlauch" auf. Das Ziel besteht darin das Element mit dem Rang pN zu finden. Mit dem Epsilon ist man zufrieden wenn man ein Element x mit $x \in [pN - \epsilon N; pN + \epsilon N]$ findet
delta	Double	Nein	Delta steuert die Versagenswahrscheinlichkeit, wobei die Werte von Null kein Versagen bis Eins nur Versagen gehen.
maxValue	Integer	Nein	Dieser Wert bestimmt den größten Wert, den die Datenstruktur verarbeiten kann. Je größer dieser Wert ist, desto mehr Speicher wird benötigt.
windowSize	Integer	Ja	Hiermit wird die maximale Anzahl der Elemente im Sliding Window bestimmt.

numberOfBuckets	Integer	Ja	Dieser Wert bestimmt die Anzahl der Buckets im Sliding Window. Ein kleiner Wert bedeutet einen geringeren Speicherverbrauch, hat aber auch eine geringere Genauigkeit zur Folge.

6.2.6 Sum Quantiles

Das Verfahren Im Rahmen der Projektgruppe wurde das *Sum Quantiles* Verfahren entwickelt. Dabei handelt es sich um ein recht einfaches aber effektives Verfahren, für die bestimmung von Quantilen. Es arbietet in einer Sliding-window Umbebung. Der Speicherverbrauch ist linear in der Grösse des Sliding-window. Es arbeitet auf ganzen Zahlen.

Das Sliding-window wird in Buckets unterteilt. Wird nun ein Element zum lernen an den Algorithmus geschickt, wird es an den jüngsten Bucket gegeben. Dieser erhört dann einen Zähler für dieses Element. Für jeden Bucket wird also protokoliert, wie oft ein Element gesehen wurde. Ausserdem wird die Anazhl der insgesamt gesehenen Elemente, und das grösste gesehene Element festgehalten. Ist der jüngste Bucket gefüllt, so wird ein neuer erstellt. Ist das Sliding-Window voll, so wird der älteste Bucket gelöscht.

Möchte man nun das ϕ-Quantil berechenen, so summiert man die Anzahl der Elemente in den Buckets auf, und bestimmt das Maximum der jeweils grössten Elemente in den Buckets. Die Summe der Anzahlen wird nun mit ϕ multipliziert und bestimmt den Rang den das Quantil hat. Unser Ziel ist es, so lange die Anzahlen von Elementen auf zu summieren, bis diese grösser gleich dem Rang ist. Dabei gehen wir die Elemente von Null bis bis zum grössten gesehenen Element aller Buckets durch, und fragen die Buckets nach der Anzahl eines Elementes i. Wenn die Summe aller Elemente von Null, bis i grösser gleich dem Rang ist, ist i das gesuchte Quantil.

Implementierung Die Implementierung ist recht einfach gehlten. Das einzige Problem, ist die Verwaltung der Zähler, in den Buckets. Hier wird eine *HashMap* verwendet, die als Schüssel, die zu einem *String* konvertierte Element enthält. Diese werden dann auf einen Zähler abgebildet. Beim lernen wird erstmal geprüft ob das Element bereits in der *HashMap* vertreten ist. Wenn ja, wird der Zähler geholt, inkrementiert, und zurückgeschrieben. Ist das Element noch nicht in der *HashMap* verzeichnet, wird es hinzugefügt und der Zählwert auf Eins gesetzt. Bei einer Anfrage für die Anzahl

eines Elements, wird der Zähler zurückgegeben, bzw. Null falls die *HashMap* keinen Eintrag für das Element enthält.

Knotenklasse

edu.udo.cs.pg542.util.node.learner.Quantiles

Eingaben

Schlüssel	Typ	Bedeutung
event:learner:input	Double	Ein neues Element, dass in die Datenstruktur eingepflegt bzw. gelernt werden soll.

Ausgaben

Schlüssel	Typ	Bedeutung
event:learner:output	Selective-Description-Model	Das Model enthält das gelernte Quantil.

Parameter

Schlüssel	Typ	Opt.	Bedeutung
windowSize	Integer	Ja	Hiermit wird die maximale Anzahl der Elemente im Sliding Window bestimmt.
numberOfBuckets	Integer	Ja	Dieser Wert bestimmt die Anzahl der Buckets im Sliding Window. Ein kleiner Wert bedeutet einen geringeren Speicherverbrauch, hat aber auch eine geringere Genauigkeit zur Folge.

Algorithmus	ϵ	δ	Fenster- / Blocklänge	Anzahl Buckets / Blöcke
Simple Quantiles	0.05	—	—	—
GK Quantiles	0.05	—	—	—
Window Sketch Quantiles	0.05	—	75.000	—
Ensemble Quantiles	0.05	—	10.000	5
Random Subset Sums	0.3	0.5	1.000	9
Sum Quantiles	—	—	100.000	10

Tabelle 36: Parametrisierung der Quantilalgorithmen

6.2.7 Evaluation

Datensatz Der Datensatz für die Evaluation der Quantilalgorithmen wurde mit dem Programm *RapidMiner*[3] erstellt. Es wurde der Datengenerator mit der Zielfunktion *random* verwendet. Die 100.000 Zufallszahlen sind reelle Zahlen aus dem Bereich $[1.0; 100.0]$.
Motiviert durch den Anwendungsfall *Web Anomaly Detection System* wurden die Quantilalgorithmen noch mit einem weiteren Datensatz evaluiert. Das Modell *Attribute Length* überwacht die Länge von Parametern und versucht anhand dieser Information Angriffe wie *Buffer-Overflow* oder *SQL-Injection* zu erkennen. Da relevante Parameter wie Benutzernamen und Passwörter in den seltensten Fällen länger als 15 Zeichen sind, wurden für diesen Datensatz 100.000 Zufallszahlen aus dem Wertebereich $[5.0; 15.0]$ generiert. Da die Ergebnisse allerdings vergleichbar sind, soll im Folgenden nur die Evaluation mit dem Datensatz mit größerem Wertebereich erläutert werden.

Parametrisierung der Algorithmen Für die Evaluation wurden die Quantilalgorithmen wie in Tabelle 36 beschrieben parametrisiert. Als Kriterium der Update-Methode von *Ensemble Quantiles* wird die Abweichung vom aktuellsten Modell verwendet. Es wird immer derjenige Block ersetzt, der am stärksten vom aktuellen Block abweicht.
Wegen des randomisierten Charakters von *Random Subset Sums* liegt die Fehlerschranke mit $\epsilon = 0.3$ höher als bei den deterministischen Verfahren. Hier hat sich gezeigt, dass durch eine Verringerung von ϵ keine signifikanten Verbesserungen der Ergebnisse resultieren. Andererseits verschlechtert sich mit sinkendem ϵ der Durchsatz, sodass wir mit $\epsilon = 0.3$ einen guten Kompromiss zwischen Güte der geschätzten Quantile und der Verarbeitungsgeschwindigkeit des Algorithmus gefunden haben.

[3] http://rapid-i.com

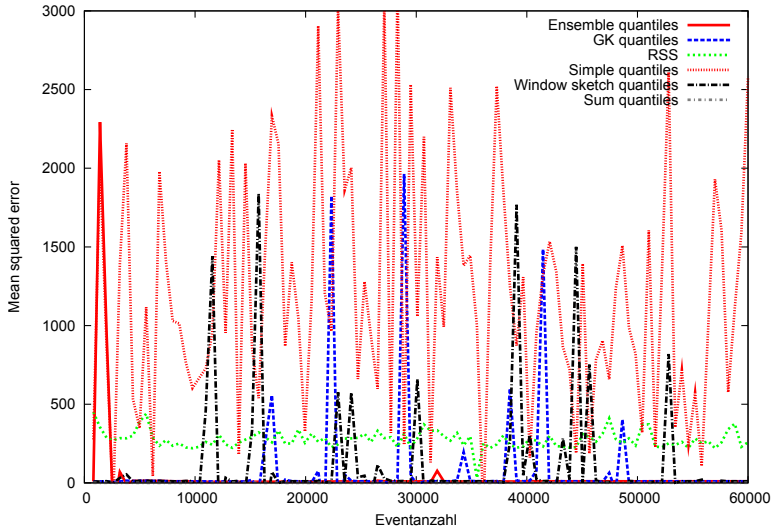


Abbildung 63: mittlerer quadratischer Fehler über den gesamten Datensatz

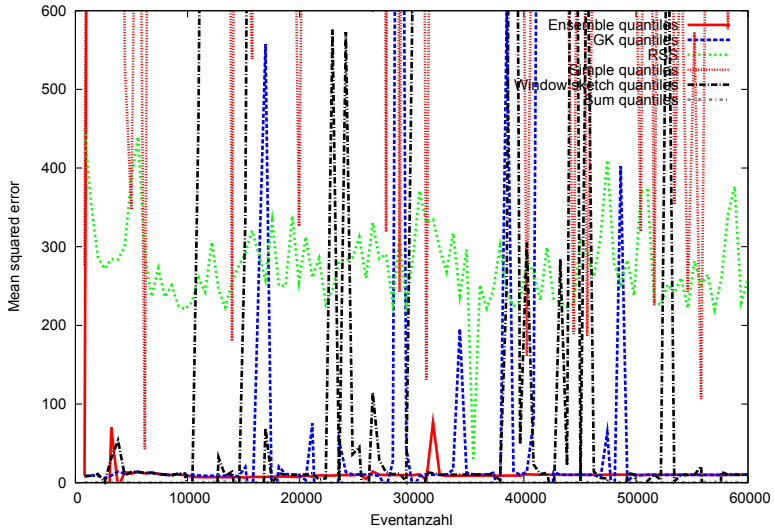


Abbildung 64: mittlerer quadratischer Fehler für die ersten 10.000 Beispiele

Quality Für die Evaluation der Güte des Algorithmus wurden durch Abfragen von ϕ-Quantilen Vektoren erzeugt. Die Vektoren erhalten die folgenden Quantile:

$$[0.25; 0.5; 0.75]$$

Ein solcher Vektor wird in jedem Schritt der Evaluation für die Algorithmen *Random Subset Sums, Simple Quantiles, GK Quantiles, Window Sketch Quantiles, Sum Quantiles* und *Ensemble Quantiles* erstellt. Außerdem wird mit *Exact Quantiles* die Wahrheit verwaltet. Als Gütekriterium wird die Abweichung des Vektors von den Exact Quantiles Ergebnissen per mittleren quadratischen Fehler (siehe Tabelle 2) bestimmt.

Bei Betrachtung der Abbildungen 63 und 64 fällt auf, dass zu Beginn der Evaluation ist der Fehler sämtlicher Quantilalgorithmen mit Ausnahme von *GK Quantiles* und *Sum Quantiles* hoch. Mit Ausnahme von *Simple Quantiles*, das keine wirkliche Datenstruktur aufbaut, sondern statische Ränge für gleiche Quantile liefert, pendelt sich der Fehler um einen Wert ein. Während der Laufzeit schlägt der Fehler für alle Verfahren von Zeit zu Zeit mit einem *Peak* aus. Die Häufigkeit und Größe dieser Peaks unterscheidet sich deutlich zwischen den Verfahren.
Während bei den deterministischen Verfahren *GK Quantiles, Window Sketch Quantiles* und Ensemble Quantiles sich die Ausschläge auf einem relativ konstant Niveau befinden, haben das Simple Quantiles Verfahren und das randomisierte Verfahren Random Subset Sums deutlich größere Peaks. Bei Random Subset Sums ist dieses Verhalten mit dem randomisierten Charakter zu erklären. Das Verfahren versagt mit Wahrscheinlichkeit δ und bestimmt in diesen Fällen sehr schlechte Quantile. Simple Quantiles erzielt hier schlechte Ergebnisse, da die wirklichen Werte der betrachteten Stichprobe nicht in die Quantilbestimmung mit einfließen. *Sum Quantiles* hingegen erreicht die bessten Ergebnisse mit einem maximalen quadratischen Fehler von 4, der sehr seltend ist. Dabei handelt es sich lediglich um Rundungsungenauigkeiten.

Die beiden Algorithmen *Window Sketch Quantiles* und Ensemble Quantiles verwenden *GK Quantiles* zur Erstellung von Zusammenfassungen des in Blöcke eingeteilten Datenstroms. Hier ist es auffällig, dass diese Verfahren zwar mit einem hohen Fehler zu Beginn des Datenstroms starten, den Fehler von GK Quantiles nach Verarbeitung von ca. 14.000 Elementen aber im Mittel deutlich unterbieten. Besonders stabil erscheint sich Ensemble Quantiles zu entwickeln. Diese Beobachtung lässt sich auf die Update-Strategie von GK Quantiles zurückführen. Das Verfahren behält alte Elemente bevorzugt in seiner Datenstruktur, da die Kapazität zum Aufnehmen weiterer Elemente mit n steigt. Hier dämpfen die beiden anderen Verfahren den auftretenden Fehler, da sie die Instanzen von GK Quantiles nur für eine feste Anzahl von Elementen verwenden und anschließend eine Zusammenfassung der GK Quantiles Datenstruktur erstellen.

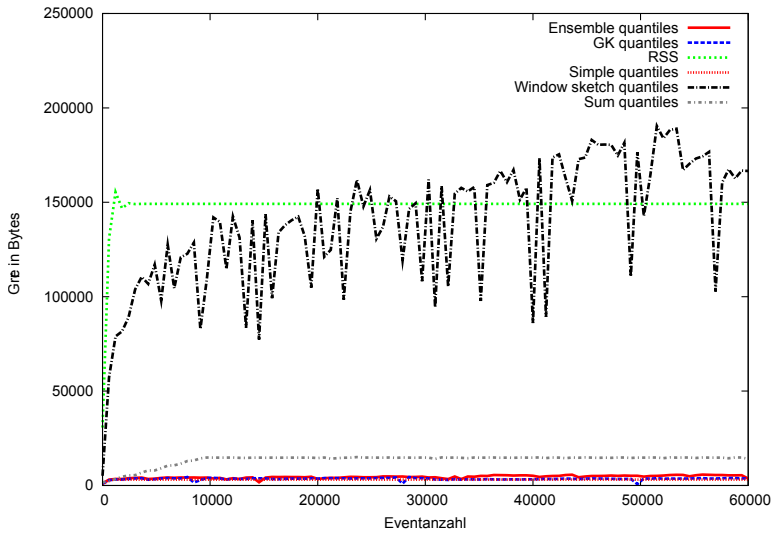


Abbildung 65: Verlauf des Speicherverbrauchs der Quantilalgorithmen

Speicher Bei der Evaluation des Speicherbedarfs hat sich gezeigt, dass mit Ausnahme von Window Sketch Quantiles alle Algorithmen einen relativ kostanten Speicherverbrauch besitzen (siehe Abbildung 65. Aufgrund des Verschmelzverhaltens von GK Quantiles schwankt der Speicher von GK Quantiles und Ensemble Quantiles etwas. Dieses Verhalten ist auch bei Window Sketch Quantiles zu beobachten, allerdings ist die Schwankung hier deutlich ausgeprägter. Da der Window Sketch Quantiles Algorithmus allerdings multiple Instanzen von GK Quantiles verwendet und mehrere Kopien aller Elemente des Datenstroms in einem *Sliding Window* verwaltet ist dieser Unterschied zu erwarten gewesen.
Die Verfahren Random Subset Sums, Sum Quantiles und Simple Quantiles reservieren nach Initialisierung einen festen Speicherbereich. Dieser Bereich wird im weiteren Verlauf nicht vergrößert oder verkleinert, wodurch sich der konstante Verlauf in Grafik 65 erklären lässt.

6.3 Klassifikation

Klassifizierung und Clustering (siehe Kapitel 6.5) beschäftigen sich mit der Gruppierung von Objekten zu Teilmengen ähnlicher Objekte. Der Unterschied zwischen den beiden Ansätzen besteht darin, dass Klassifizierungsverfahren auf gelabelten Daten arbeiten, wohingegen Clustering ungelabelte Daten zu Gruppen zusammenfasst.

Im Rahmen der Projektgruppe wurden die Klassifizierungsverfahren Naive Bayes, Hoeffding Bäume, Very Fast Decision Trees und eine OnlineSVM implementiert.

6.3.1 Naive Bayes

Idee Der *Naive Bayes* Klassifizierer [26] nutzt das Bayestheorem zur Klassifikation von Daten. Bayestheorem und Naive Bayes Klassifizierer sind nach ihrem Entdecker, dem englischen Mathematiker Thomas Bayes, benannt. Das Verfahren ordnet jeden Datenpunkt der Klasse zu, für die die Wahrscheinlichkeit, dass der Datenpunkt zu der Klasse gehört, maximal ist. Dadurch ist es möglich mit diesem Klassifizierer auf einfache Weise und damit extrem schnell eine Klassifizierung durchzuführen.

Der Ansatz basiert auf der Annahme, dass alle Attribute eines Datenpunktes unabhängig voneinander sind und nur vom Klassenattribut abhängen. Trotz dieser - in der Praxis häufig nicht zutreffenden - Annahmen erzeugt ein Naive Bayes Klassifizierer bei Anwendungen in der Praxis oft eine sehr gute Klassifikation. Eklatante Schwächen weißt das Verfahren nur auf, wenn die Attribute sehr stark korrelieren.

Mathematisch gesehen handelt es sich bei dem Naive Bayes Kassifikator also um eine Funktion b, die jeden Datenpunkt aus dem Merkmalsraum X mit n Attributen $X = (x_1, x_2, x_3, ..., x_n)$ auf eine Klasse aus einer Menge möglicher Klassen C abbildet:

$$b : \mathbb{R}^f \rightarrow C$$

Die Anzahl der Klassen $|C|$ muss dabei mindestens zwei sein.

Der Klassifizierung liegen, wie bereits erwähnt, folgende wahrscheinlichkeitstheoretische Formeln zu Grunde:

Bayestheorem:

$$P(C_i|X) = \frac{P(X|C_i)P(C_i)}{P(X)}$$

Da es sich bei $P(X)$ lediglich um einen Normalisierungsfaktor handelt, der für alle Klassen C_i identisch ist, muss dieser nicht berechnet werden.

A-Priori Wahrscheinlichkeit einer Klasse:

$$P(C_i) = \frac{n}{m}$$

wobei n der Anzahl der bisher gesehenen Examples in der Klasse C_i und m der Anzahl aller bisher gesehenen Examples entspricht.

Umsetzung Für die Umsetzung wird davon ausgegangen, dass die Datenpunkte nur numerische Merkmale enthalten.

Der Klassifizierungs-Algorithmus liefert die wahrscheinlichste Klasse Y für einen gegebenen Datenpunkt $X = (x_1, x_2, x_3, ..., x_n)$

$$arg\max_{Y} P(Y|X) = \frac{P(Y)}{P(X)} \times \prod_{k=1}^{n} P(x_k|Y)$$

Im intuitivsten Fall müsste der Klassifizierer alle bereits gesehenen Attribute mit ihren Ausprägungen und Häufigkeiten pro Klasse speichern, um damit die wahrscheinlichste Klasse Y berechnen zu können. Dieser intuitive Ansatz kann allerdings zu Problemen führen, wenn Werte für ein Attribut auftauchen, die noch nicht gesehen wurden. Daher ist dieser Ansatz bei der vorliegenden Implemtierung von Naive Bayes für numerische Attribute nicht gewählt worden. Anstatt jeden gesehenen Wert eines Attributes mit seiner Häufigkeit zu speichern, werden nur der Mittelwert M und die Standardabweichung σ für jedes numerische Attribut gespeichert. Um die Standardabweichung und das arithmetische Mittel kontinuierlich aktualisieren zu können wurde folgende Formeln verwendet:

$$\sigma^2(1) = 0, \sigma^2(k) = \sigma(k-1) + (x(k) - M(k-1)) * (x(k) - M(k))$$

$$\sigma(k) = \sqrt{\sigma^2(k)/n - 1}$$

$$M(1) = 0, M(k) = M(k-1) + \frac{x(k) - M(k-1)}{n}$$

wobei n die Anzahl der bisher gesehenen Beispiele ist.

Zusätzlich wird angenommen, dass die Datenpunkte eines Attributes normalverteilt sind. Mit Hilfe der Dichtefunktion der Normalverteilung, des berechneten Mittelwertes und der berechneten Standardabweichung kann dann das Problem der fehlenden Werte umgangen werden.

Diese Vorgehen funktioniert allerdings nur bei numerischen Attributwerten. Um mit nominalen Attributen umgehen zu können muss auf eine alternative Berechnung z.B. nach dem zuvor erwähnten Standardverfahren zurück gegriffen werden. Um nominale Attribute zu unterstützen wurde dieses Standardverfahren implementiert. Daher kommt es bei nominalen Attributen allerdings zu Problemen, bei fehlenden Werten.

Knotenklasse

edu.udo.cs.pg542.util.node.learner.NaiveBayesNode

Eingaben

Schlüssel	Typ	Bedeutung
event:learner:input	LabeledExample	Ein gelabeltes Example, auf dem gelernt werden kann

Ausgaben

Schlüssel	Typ	Bedeutung
event:learner:output	Integer	Der Index der Klasse, der für das Example berechnet wurde

Parameter

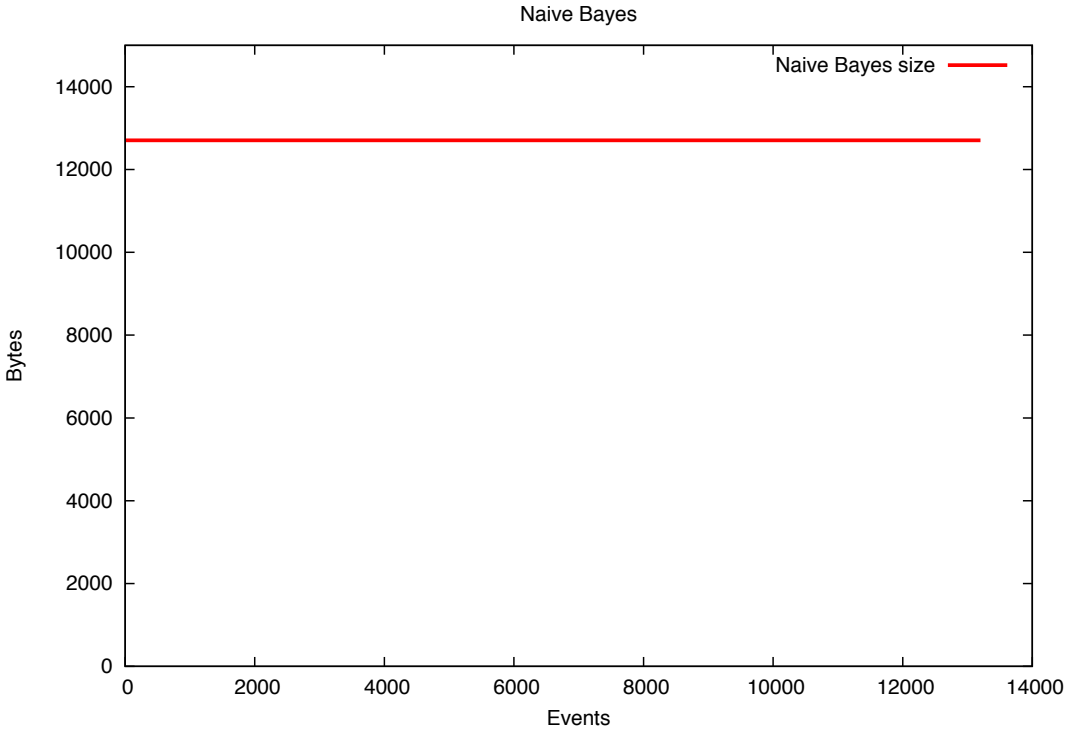
Schlüssel	Typ	Opt.	Bedeutung
features	List<String>	Nein	Eine Liste von Featurekeys, wie sie in dem Example übergeben werden
classCount	Integer	Nein	Anzahl der verwendeten Klassen
threadSafe	Boolean	Ja	Aktivierung von Thread sicheren Datenstrukturen {default: true}
defaultParameterType	FeatureType	Ja	Standard Feature Typ: {NOMINAL, NUMERIC} {default: NUMERIC}
nonDefaultTypeParameters	List<String>	Ja	Eine Liste der Featurekeys, deren Typen vom default Typ abweichen

Evaluation Um die Streamvariante des Naive Bayes Algorithmus zu Evaluieren, wurden verschiedene uci Datensätze genutzt. Bei den für die Grafiken verwendeten Daten handelt es sich um den *letter recognition* Datensatz[4]. Dieser enthält 16 nummerische Attribute, 26 Klassen und 20.000 Beispiele. Um Vergleichswerte für die unterschiedlichen Gütemaße zu haben, wurde der gleiche Datensatz mit dem Naive Bayes Batch Verfahren aus RapidMiner[5] analysiert und verglichen.

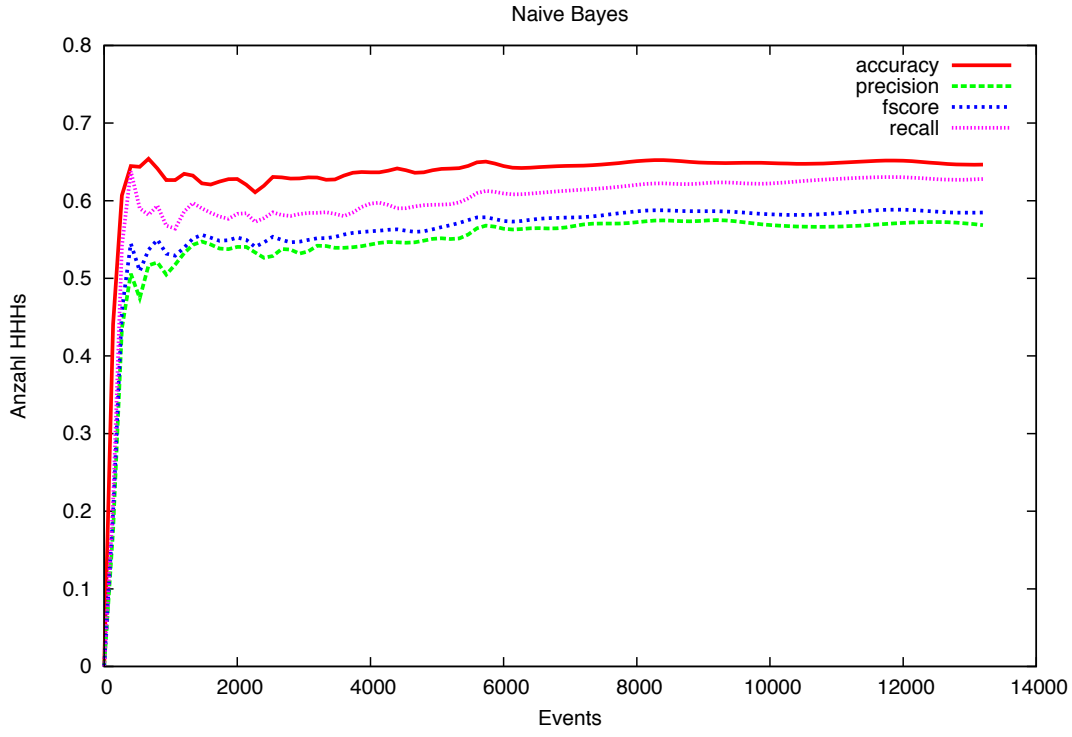
Dabei zeigte sich, dass die Ergebnisse des hier implementierten Stream Algorithmus sehr nah an die Ergebnisse des Batchverfahrens heran kommen. Während die Batch Variante eine accuracy von 64,62% erreicht, erzielt der Stream Algorithmus im Mittel 63%. Auf einem weiteren Testdatensatz erreichten beide Lernverfahren sogar die gleiche accuracy von 99,6%. Die Nutzung der Dichtefunktion der Normalverteilung in Kombination mit dem Mittelwert und der Standardabweichung zeigt sich somit als extrem effizient für nummerische Attribute.

Wie zuvor erwähnt werden nur der Mittelwert und die Standardabweichung pro Kombination aus Attribut und Klasse gespeichert. Dadurch ergibt sich ein sehr niedriger und konstanter Speicherverbrauch. Für den letter regognition Datensatz liegt dieser bei gerade einmal 12.706 Byte.

[4]http://archive.ics.uci.edu/ml/datasets/Letter+Recognition
[5]http://rapid-i.com



(a) Speicherverbrauch des Naive Bayes Stream Algorithmus



(b) Qualität des Naive Bayes Stream Algorithmus

Abbildung 66: Evaluierung des Naive Bayes Stream Algorithmus auf 20.000 Beispielen, des uci *letter recognition* Datensatzes

6.3.2 Hoeffding Bäume

Idee Der Hoeffding-Tree-Algorithmus wie er als Basiskonzept von VFDT in [15] vorgestellt wird, ermöglicht das Lernen von Entscheidungsbäumen mit einer gegebenen Fehlerschranke auf zeitlich konstanten Streams ohne *concept drift*. Es können nur nominale *Attribute* (auch: *Features*) behandelt werden, numerische können jedoch prinzipiell per einfacher, statischer Diskretisierung konvertiert werden.
Als TDIDT (*Top-Down Induction of Decision Trees*) inspiriertes Verfahren wächst das erstellte Baum-Modell indem sukzessiv Blätter zu inneren Knoten werden, die eintreffende Beispiele anhand der Werte eines Attributs aufteilen. Welche Attribute zur Aufteilung gewählt werden, entscheidet das benutzer-definierte Gütekriterium.

Umsetzung Als Gütekriterien sind *Entropie* oder *Gini-Index* möglich. Der *Gewinn* $G(.)$ eines Attributes f in einem Blatt ist definiert als *Güte ohne Teilung - Güte mit Teilung an* f. Wird *Entropie* als Gütekriterium gewählt, so wird der Gewinn als *Information Gain* bezeichnet.

Hoeffding-Schranke (oder additive Chernoff-Schranke) Die wesentliche Idee des Verfahrens basiert darauf, dass mit Wahrscheinlichkeit $1-\delta$ der *wahre* Gewinn - über den gesamten Stream - eines Attributes f zu jedem Zeitpunkt t nicht kleiner ist als $G_t(f) - \epsilon$ mit $\epsilon = \sqrt{\frac{R^2 ln(1/\delta)}{2n}}$. Wobei R der höchstmögliche Gewinn und n die Anzahl der bis t gesehenen Beispiele ist.
Mit diesem Resultat ist es möglich Entscheidungsbäume aus Streams zu konstruieren, die mit großer Wahrscheinlichkeit identisch sind zu jenen, die von einem Batch-Lerner konstruiert würden, der den gesamten Stream zur Verfügung hätte.

Algorithmus Im Initialisierungsschritt wird die Baumwurzel als Blatt erstellt und für dieses Blatt alle Zähler auf 0 gesetzt und die *verbleibenden Attribute* als die Menge aller Attribute definiert.

Eintreffende Beispiele werden zunächst gemäß des bisher gelernten Baumes in ein Blatt b sortiert. Dort wird für jedes in b verbleibende Attribut f ein Zähler um eins inkrementiert. Die Zähler repräsentieren die Anzahl der in b gesehenen Beispiele, die für f den Wert v und das *Label* l hatten. Nachdem die korrespondierenden Zähler für ein Beispiel erhöht wurden, wird das Beispiel verworfen.
Sei F die Anzahl der in b verbleibenden Attribute, V_f die Anzahl möglicher Werte für Attribut f und L die Anzahl möglicher Label. Dann müssen für das Blatt b konstant $L * \sum_{f=1}^{F} V_f$ Zähler gehalten werden. Der Speicherplatzbedarf ist somit unabhängig von der Anzahl der gesehenen Beispiele.

Nachdem in einem Blatt b die Zählerstände inkrementiert wurden, wird b mit dem häufigsten Label (über alle in b gesehenen Beispiele) versehen. Anhand der neuen

Zählerstände wird für jedes in b verbleibende Attribut f der *Gewinn* errechnet, den man erhält, wenn b zu einem inneren Knoten wird, der anhand der Werte von f ankommende Beispiele weiter aufteilt.

Wenn

- nicht alle gesehenen Beispiele dasselbe Label haben und
- genügend Beispiele gesehen wurden, sodass die *Hoeffding-Schranke* garantiert, dass - über den gesamten Stream - Attribut f den höchsten Gewinn an diesem Blatt liefert (d.h. für jedes Attribut $h \neq f : G(f) - G(h) > \epsilon$) und
- der Gewinn von f einen Minimalwert übersteigt (d.h. $G(f) - mingewinn \geq \epsilon$)

, wird das Blatt zu einem inneren Knoten, der nachfolgende Beispiele an f testet. Für jeden Wert von f wird ein neues Kind eingefügt, dessen Zähler mit 0 initialisiert werden. Aus der Menge der verbleibenden Attribute wird f für jedes Kind entfernt.

Implementierungsdetails des Knotens Im Paket *edu.udo.cs.pg542.util.node.learner* befindet sich der *HoeffdingTreeLearningNode*, der in einem Knotennetzwerk einen Hoeffding-Baum lernt. Der Lerner erwartet eine initiale Menge von Beispielen in einem *SignalEvent*, die sämtliche im späteren Lernverlauf möglichen Attribute, Attributwerte (ausgenommen numerische) und Label enthält. Erst nach der dann folgenden Initialisierung verarbeitet der Knoten ankommende Beispiele. Ferner darf der Stream keiner zeitlichen Schwankung (concept drift) unterliegen, damit die Aussage der Hoeffding-Schranke gilt.

Der Vorverarbeitungsknoten sammelt die zur Initialisierung notwendige Menge von Beispielen und leitet im Anschluss nur kompatible Beispiele an den Lerner weiter. Zusätzlich randomisiert er per Cache den Stream um einen periodischen concept drift auszugleichen.

Zusätzliche Variablen pro Blatt um die Geschwindigkeit zu erhöhen

- Die totale Anzahl der im Blatt gesehenen Beispiele: Wird an vielen Stellen benötigt.
- Das Attribut mit den geringsten verschiedenen Werten: Einige Berechnungen (z.B. häufigstes Label oder Gewinn ohne Teilung) benötigen nur die Zähler eines einzigen, beliebigen Attributes. In diesem Fall kann ein Geschwindigkeitsgewinn erreicht werden, indem man über das Attribut mit den wenigsten Zählern (Attributwerten) iteriert.
- Information ob alle bisherigen Beispiele dasselbe Label hatten: Wird pro neuem Beispiel einmal abgefragt.

- Das zuletzt berechnete häufigste Label und die Zahl neuer Beispiele, bis das häufigste Label neu berechnet werden muss: Pro Beispiel wird gefragt, welches Label nun das häufigste ist. Diese Variablen verhindern eine ständige Neuberechnung, wenn ein Label mit Abstand am häufigsten vorkam. Treffen weitere Beispiele mit dem zuletzt berechneten häufigsten Label ein, wird die Anzahl der Beispiele bis zum Update erhöht, sonst verringert. Hierbei wird angenommen, dass die *equals*-Methode der Labelklasse *schnell* das Ergebnis liefert, was praktisch immer der Fall ist.

Knotenklasse

edu.udo.cs.pg542.util.node.learner.HoeffdingTreeLearningNode

Eingaben

Schlüssel	Typ	Bedeutung
event:learner:input	LabeledExample	Die Domäne der eingehenden Beispiele muss konstant über den gesamten Stream sein

Ausgaben

Schlüssel	Typ	Bedeutung
event:learner:output	HoeffdingTree-Model	Repräsentiert einen Entscheidungsbaum

Parameter

Schlüssel	Typ	Opt.	Bedeutung
default-type	String	Ja	Parameter für den Defaulttyp von Features - 'nominal' oder 'numeric'. Bei nicht gesetztem Parameter werden die Featuretypen automatisch bestimmt.
not-default-features	List<String>	Ja	Definiert alle Features mit inversem Typen zum Default-Type

min-different-values	Integer	Ja	Wenn der 'default-type' automatisch bestimmt wird, dann gibt dieser Parameter die Mindestanzahl von verschiedenen Werten pro Feature an, die mindestens existieren müssen, damit das Feature als numerisch erkannt wird.
quantiles	List<Double>	Ja	Die hier angegebenen Quantile diskretisieren numerische Werte, die dann als Splitpunkte verwendet werden.
delta	Double	Ja	Die Fehlerschranke des Algorithmus
quality-criterion	String	Ja	Bestimmt das zu benutzende Qualitätskriterium. Entweder 'information-gain' oder 'gini-index-gain'.
min-quality-for-split	Double	Ja	Dieser Parameter dient zum Prepruning. Es bestimmt den minimale Qualitätsgewinn, den ein Split liefern muss.

6.3.3 VFDT

Idee *Very Fast Decision Tree (VFDT)* basiert auf dem in 6.3.2 vorgestellten Hoeffding Baum. Es verbessert punktuell dieses Verfahren.

Umsetzung In der vorliegenden Implementierung wurden nur die unter *Geschwindigkeitsgewinn* erwähnten Verbesserungen umgesetzt.

- **Geschwindigkeitsgewinn**
  - **Unentschieden**: Falls der Unterschied des Gewinns zweier Attribute sehr klein ist, werden sehr viele Beispiele (und viel Zeit) benötigt, bis ϵ hinreichend klein wird um sich für ein Attribut entscheiden zu können. Ist $\epsilon < \tau$ entscheidet VFDT auf Unentschieden zwischen den zwei Attributen und das Blatt wird an dem derzeit besten Attribut geteilt. Der Schwellwert τ ist vom Benutzer vorzugeben.

- $G(.)$ **Neuberechnung**: Die Neuberechnung des Gewinns für jedes Attribut pro eintreffendem Beispiel benötigt die meiste Zeit. Es ist jedoch unwahrscheinlich, dass die Frage an welchem Attribut ein Blatt geteilt wird, durch das jeweils zuletzt ankommende Beispiel entschieden werden kann. VFDT wartet daher eine benutzer-definierte Anzahl von Beispielen n_{min} in einem Blatt ab, bevor der Gewinn für jedes Attribut neu berechnet wird.

- **Weniger Speicherverbrauch**
  Die hier erwähnten Punkte können das Ergebnis verschlechtern. Es wird angenommen, dass genügend Speicher vorhanden ist.

  - **Blätter deaktivieren**: Der Speicher für ein Blatt kann freigegeben werden, wenn das Teilen dieses Blattes die Vorhersagegenauigkeit des gesamten Baums zu gering erhöhen würde. Die Nützlichkeit eines Blattes b wird definiert als $p_b * e_b$, wobei p_b die Wahrscheinlichkeit ist, dass ein beliebiges Beispiel in b fällt und e_b die Fehlerrate in b ist. Wird ein Blatt deaktiviert, wird für dieses zukünftig nur $p_b * e_b$ aktualisiert. Sollte dieser Wert im Vergleich zu den restlichen Blättern wieder hoch genug sein, wird das Blatt reaktiviert.

  - **schwache Attribute**: Wenn der Gewinn eines Attributs kleiner wird als $G(f) - \epsilon$, wobei f das Attribut mit dem höchsten Gewinn ist, kann das *schwache* Attribut verworfen und die korrespondierenden Zähler freigegeben werden.
    Je nach Speicherstruktur kann das Entfernen beliebiger Attribute sehr kostenintensiv sein, was in der vorliegenden Implementierung der Fall ist.

- **Initialisierung**
  Um auf kleinen Beispielmengen keinen Nachteil gegenüber Batch-Lernern zu haben, kann ein von einem Batch-Lerner konstruierter Baum zur Initialisierung genutzt werden (statt einer leeren Wurzel), der dann erweitert wird.
  Unter der Annahme, dass auf (hochfrequenten) Streams gelernt wird, ist dieser Schritt nicht notwendig.

- **Rescan**
  Jedes Beispiel wird nur zur Teilung genau eines Knotens benutzt. Batch-Lerner betrachten pro Knoten sämtliche Beispiele. Um diesen Nachteil zu relativieren, können Beispiele mehrfach zum lernen (in verschiedenen Knoten) benutzt werden.
  Solange Beispiele in ausreichender Anzahl eintreffen, ist ein *Rescan* nicht notwendig. Bei hochfrequenten Streams ist hierfür keine Zeit vorhanden.

Der Knoten, *VFDTLearningNode* erbt von *HoeffdingTreeLearningNode*. Insofern gelten sämtliche unter 6.3.2 beschriebenen Hinweise in gleicher Form.

Knotenklasse

edu.udo.cs.pg542.util.node.learner.VFDTLearningNode

Eingaben

Schlüssel	Typ	Bedeutung
event:learner:input	LabeledExample	Die Domäne der eingehenden Beispiele muss konstant über den gesamten Stream sein

Ausgaben

Schlüssel	Typ	Bedeutung
event:learner:output	HoeffdingTree-Model	Repräsentiert einen Entscheidungsbaum

Parameter

Schlüssel	Typ	Opt.	Bedeutung
n-min	Integer	Ja	Die Anzahl der Beispiele, die in jedem Blatt abgewartet werden, bevor der Qualitätsgewinn neu berechnet wird
tau	Double	Ja	Fällt ϵ unter diesen Schwellwert, wird zwischen den zwei besten Attributen eines Blattes ein Unentschieden angenommen und eine Entscheidung erzwungen
default-type	String	Ja	Parameter für den Defaulttyp von Features - 'nominal' oder 'numeric'. Bei nicht gesetztem Parameter werden die Featuretypen automatisch bestimmt.
not-default-features	List<String>	Ja	Definiert alle Features mit inversem Typen zum Default-Type

min-different-values	Integer	Ja	Wenn der 'default-type' automatisch bestimmt wird, dann gibt dieser Parameter die Mindestanzahl von verschiedenen Werten pro Feature an, die mindestens existieren müssen, damit das Feature als numerisch erkannt wird.
quantiles	List<Double>	Ja	Die hier angegebenen Quantile diskretisieren numerische Werte, die dann als Splitpunkte verwendet werden.
delta	Double	Ja	Die Fehlerschranke des Algorithmus
quality-criterion	String	Ja	Bestimmt das zu benutzende Qualitätskriterium. Entweder 'information-gain' oder 'gini-index-gain'.
min-quality-for-split	Double	Ja	Dieser Parameter dient zum Prepruning. Es bestimmt den minimale Qualitätsgewinn, den ein Split liefern muss.

Evaluation Für die Evaluation von VFDT wurde der einfache, insbesondere zum Testen von Entscheidungsbäumen ausgelegte, UCI Datensatz *car evaluation*[6] herangezogen.

Datensatz
Der Datensatz ordnet Autos in Abhängigkeit von 6 ausschließlich nominalen Attributen (Anschaffungspreis, Unterhaltskosten, Anzahl Türen, Anzahl Personen, Größe Kofferraum, Sicherheit) in 4 Klassen ein: inakzeptabel, akzeptabel, gut, sehr gut.
Der Datensatz beinhaltet 1728 Beispiele ohne fehlende Werte.
Häufigste Klasse: inakzeptabel mit 70.023 %

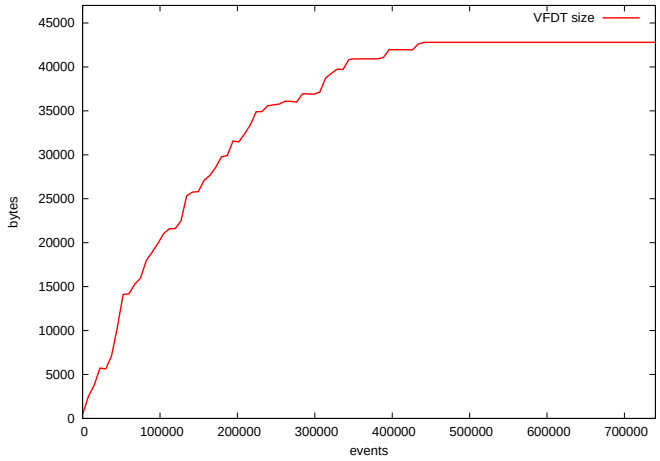
Parametereinstellungen
Da 1728 Beispiele in diesem Fall zu wenig sind um einen Aussagekräftigen Entscheidungsbaum zu produzieren, wurde der Datensatz immer wieder neu gelesen. Der Stream bestand somit aus einer unendlichen Aneinanderreihung der immergleichenen Beispiele. Zur Evaluation wurde die *Holdout*-Methode benutzt, die in diesem Fall

[6] http://archive.ics.uci.edu/ml/datasets/Car+Evaluation

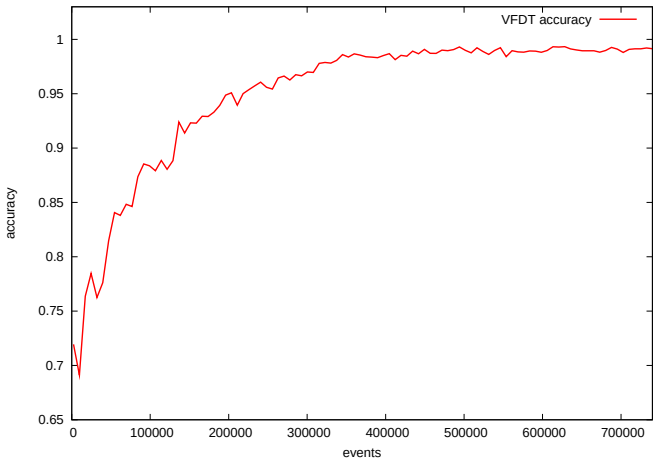
den Stream wie folgt aufteilte: 66 % zum Lernen und 34 % zum Testen. In Kombination mit den unendlichen Iterationen über den selben Daten, testet man jedoch letztlich auf den Trainingsdaten.
Für den Lerner wurden die Default-Parameterwerte gesetzt:

- $\delta = 0.0000001$
- Qualitätskriterium = information gain
- Mindestqualität für Split = 0 (Prepruning an)
- $n_{min} = 200$
- $\tau = 0.05$

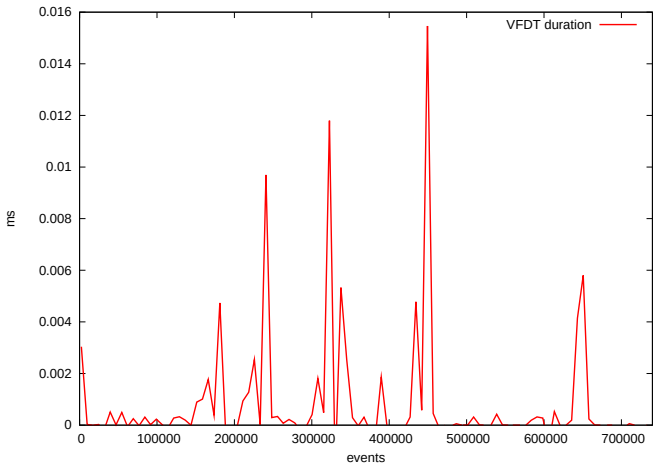
Ergebnisse
Für die Messungen wurde die Aktualisierungsrate auf 5 Sekunden gestellt. Die Messwerte bilden somit jeweils den Mittelwert aus einem 5 Sekunden Intervall.
Aus den Abbildungen 67a und 67b erkennt man deutlich das Prinzip des Verfahrens. Jeder Anstieg in Abb. 67a repräsentiert das Hinzufügen von weiteren Blättern, wenn ein Splitattribut gefunden wurde, und somit eine Verfeinerung des Modells. Hiermit einhergehend ist ein Anstieg der Accuracy in Abb. 67b. Wurden erst wenige Beispiele verarbeitet, liegt die Accuracy bei ca. 0,7. Dieser Wert war zu erwarten, da solange das Modell nur aus der Wurzel besteht, lediglich die häufigste Klasse („inakzeptabel“ bei 70,023 % aller Beispiele) vorhergesagt wird. Bei einem Vergleich der Kurven aus beiden Abbildungen, wird deutlich, dass mit jeder Verfeinerung des Modells auch die Accuracy steigt. Nach ca. 450.000 verarbeiteten Beispielen stellt sich zunächst ein stabiler Zustand des Modells ein. Die Accuracy liegt dort bei ca. 99 %. Lässt man das Experiment weiter laufen, verfeinert sich das Modell später ab ca. 2.200.000 Beispielen auf dann 50.000 Byte und erreicht eine Accuracy von genau 100 %. Dieses Ergebnis ist angesichts des Datensatzes und der Tatsache, dass auch auf den Testdaten trainiert wird nicht verwunderlich.
Wie aus Abb. 67c hervorgeht, liegt die Zeit die pro Beispiel zum Lernen verwendet wird maximal im Bereich von wenigen Nanosekunden (meist $< 1ns$).



(a) Speicherverbrauch in bytes



(b) Qualität (accuracy)



(c) Zeit zur Bearbeitung eines Beispiels in ms

Abbildung 67: Evaluierung des VFDT Algorithmus auf 700.000 Beispielen des UCI Datensatzes *car evaluation*

6.3.4 Perceptron

Idee Der Perceptron Lernalgorithmus ist eines derjenigen Verfahren, welche eine Hyperebene konstruieren, um die einzelnen Beispiele eines Zweiklassenproblems voneinander zu trennen. Die Trennung durch eine Hyperebene in beliebig dimensionalen Räumen erfolgt immer linear. Damit ist auch die Klasse der Probleme festgelegt, die durch eine einfache lineare Trennung gelöst werden können. Der Perceptron-Lerner basiert auf dem klasssischen Verfahren Rosenblatts [36], bei dem ein einzelnes informationsverarbeitendes Neuron zum Einsatz kommt. Angewendet wird das Prinzip der gewichteten Summe aller Attributwerte inklusive einem Verzerrungswert, dem sogenannten Bias. Jenseits aller biologischen Beschreibungen ist das Percepton ein linearer Klassifikator mit einer Diskriminanzfunktion von folgender Form:

$$f(x) = \hat{\beta} + \hat{\beta}_1 x_1 + \hat{\beta}_2 x_2 + ... + \hat{\beta}_n x_n$$

Rosenblatt konnte zeigen, das für die eben erwähnte Klasse linear separierbarer Probleme der Perceptronalgorithmus mit endlich vielen Schritten zu einer separierenden Hyperebene gelangt.

Um eine Hyperebene konstruieren zu können liegt es nahe, zunächst einen kurzen Blick auf die Darstellung von Ebenen in der linearen Algebra zu werfen. Gebildet wird eine Hyperebene H aus allen Punkten x_0 für die die Gleichung $f(x) = \beta_0 + \beta^T x = 0$ erfüllt ist. Für H ist der zugehörige Normalenvektor gegeben durch $\beta^* = \frac{\beta}{\|\beta\|}$. Damit lässt sich für die vorzeichenbehaftete Distanz eines Punktes x zu H folgende Gleichung herleiten:

$$\beta^{*T}(x - x_0) = \frac{1}{\|\beta\|}\left(\beta^T x + \beta_0\right) = \frac{1}{\|f'(x)\|} f(x)$$

Das bedeutet, dass sich der Wert von $f(x)$ proportional zur Distanz zwischen x und H verhält und die gewichtete Summe, die ja die Grundlage für $f(x)$ darstellt, ein brauchbares Distanzmaß ist.

Umsetzung Der Perceptronalgorithmus versucht eine trennende Hyperebene für das Ausgangsproblem zu finden, indem er die Kosten für fehlerhaft klassifizierte Punkte zu minimieren sucht. Unter Kosten versteht sich der Distanzwert des betreffenden Punktes zur Hyperebene. Eine Fehlklassifikation wird umso schwerwiegender, je weiter dieser Punkt von der Trennungsebene entfernt liegt. Ein Punkt mit der Vorhersage $y_i = 1$ gilt als fehlerhaft klassifiziert, wenn $x_i^T \beta + \beta_0 < 0$ und genau umgekehrt für einen Fehler bei Vorhersage von $y_i = -1$. Grundsätzlich soll folgende

Funktion minimiert werden:

$$D\left(\beta, \beta_0\right) = -\sum_{i \in M} y_i \left(x_i^T \beta + \beta_0\right)$$

Der Gradient des Minimierungsproblems kann beschrieben werden durch

$$\partial \frac{D\left(\beta, \beta_0\right)}{\partial \beta} = -\sum_{i \in M} y_i x_i,$$

und

$$\partial \frac{D\left(\beta, \beta_0\right)}{\partial \beta_0} = -\sum_{i \in M} y_i.$$

Faktisch nutzt das Perceptronverfahren das Prinzip des Gradientenabstiegs, da nach jeder Beobachtung einer Fehlklassifikation die Parameter β und β_0 wie folgt aktualisiert werden:

$$\begin{pmatrix} \beta \\ \beta_0 \end{pmatrix} \leftarrow \begin{pmatrix} \beta \\ \beta_0 \end{pmatrix} + p \begin{pmatrix} y_i x_i \\ y_i \end{pmatrix}$$

Der Parameter p wird als Lernrate bezeichnet, den er bestimmt mit welchem Gewicht die Änderungen an den Koeffizienten der Hyperebene vorgenommen werden sollen.

Knotenklasse

edu.udo.cs.pg542.util.node.learner.PerceptronLearningNode

Eingaben

Schlüssel	Typ	Bedeutung
event:learner:input	LabeledExample	Eine Folge gelabelter Daten, die als Trainingsinstanzen verwendet werden

Ausgaben

Schlüssel	Typ	Bedeutung
event:learner:output	HyperplaneModel	Das Modell einer Hyperebene, das auf Grundlage der Trainingsinstanzen erzeugt wird

Parameter

Schlüssel	Typ	Opt.	Bedeutung
kernel_type	Integer	Ja	Auswahl des Kerneltyps, der als Kernfunktion für das Training verwendet werden soll
learn_rate	Double	Ja	Der Parameter bestimmt das Gewicht mit dem jede Änderung an den Modelparametern vorgenommen wird.

6.4 Regression

Bei der Regression handelt es sich um eine Ausgleichsrechnung. Ziel dessen ist es, eine Beziehung zwischen mehreren Variablen herzustellen. Des Weiteren sollen mit Hilfe der Regressionsgeraden fehlende Werte bestimmt werden können. In der folgenden Abbildung 68, ist ein Beispiel für eine Regression zu sehen. Hierbei handelt es sich um den Sonderfall Lineare Regression. Das Regressionsverfahren erhält die einzelnen Features und das Label. Die Verteilung der einzelnen Werte ist in dem Diagramm auf der linken Seite zu erkennen. Das Ziel der Regression ist es nun eine Gerade zu berechnen, mit der die einzelnen Werte repräsentiert werden. Die Regressionsgerade ist in Abbildung 68 zu sehen.

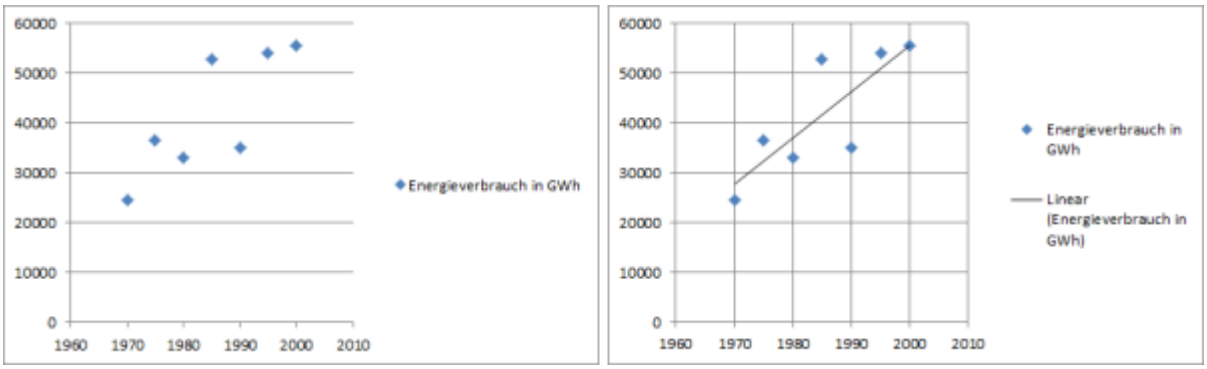


Abbildung 68: Beispiel für eine Regression

6.4.1 Lineare Regression

Idee Die Idee der linearen Regression ist es, eine Menge an Werten bestmöglich durch eine Gerade zu repräsentieren. Hierbei muss beachtet werden, dass eine maßgebende Abweichung von der Regressionsgeraden auch eine Unregelmäßigkeit in den Messwerten repräsentieren kann. Des Weiteren kann mit Hilfe der errechneten Regressionsgeraden fehlende Werte beerchnet werden.

Umsetzung Die folgenden Formeln basieren auf dem Paper [4]. Bei der linearen Regression werden Wertepaare (x_i, y_i) betrachtet. Das i repräsentiert die fortlaufenden Beispiele, die betrachtet werden. Dabei stehen die Werte x und y in einem Verhältnis zueinander. Sollen Messfehler mitbetrachtet werden, so muss die Geradengleichung um ein ε erweitert werden. Für die Gerade steht die folgende Gleichung:

$$y_i = \alpha + \beta x_i$$

Da α und β nicht durch umstellen der Gleichung berechnet werden können, müssen diese mathematisch geschätzt werden. Somit erhält man die folgende Geradengleichung:

$$y_i = a + bx_i$$

Durch die Betrachtung der Steigung der Geraden an einem Punkt kann man b wie folgt errechnen:

$$b = \frac{Kovarianz_{xy}}{Varianz_{xx}}$$

Für die Berechnung von a folgt:

$$a = \bar{y} - b\bar{x}$$

Somit erhält man die Regressionsgleichung in der folgenden Form:

$$y_i = \bar{y} - \frac{Kovarianz_{xy}}{Varianz_{xx}}\bar{x} + \frac{Kovarianz_{xy}}{Varianz_{xx}}x_i$$

Bei einem neuen Wertepaar wird betrachtet, ob es noch zur Gleichung passt. Ist dies nicht der Fall, so wird eine neue Gerade errechnet. Bei der Prüfung wird die Varianz, des neuen Wertepaares mit der Varianz des vorherigen Wertepaares betrachtet. Ist die Varianz des neuen Wertepaares kleiner, so wird eine neue Gerade errechnet. Ist dies nicht der Fall, so bleibt die vorherige Gerade erhalten. Ist der y-Wert von mehreren Faktoren abhängig, so kann eine *Multiple Lineare Regressionsgerade* zu dem dazugehörigen Wertepaar $(a_1, b_1, ..., x_1, y_1)$ errechnet werden, die wie folgt aussieht:

$$y = \alpha + \beta_1 a_i + \beta_2 b_i + ...$$

Das Verfahren bietet ebenfalls die Möglichkeit einen y-Wert zu den dazugehörigen Features zu berechnen. Bei diesem Verfahren muss beachtet werden, dass das Verfahren zwar mit mehreren Features arbeiten kann, aber nicht mit Features die mehrere Dimensionen haben.

Knotenklasse

edu.udo.cs.pg542.util.node.learner.LinearRegressionNode

Eingaben

Schlüssel	Typ	Bedeutung
event:learner:input	LabeledExample	Dies ist das Example auf dem der Learner lernt.

Ausgaben

Schlüssel	Typ	Bedeutung
event:learner:output	Linear-RegressionModel	Das *Model* enthält die Regressionsgerade.

Parameter

Schlüssel	Typ	Opt.	Bedeutung
attributeNames	List<String>	Nein	*AttributeNames* enthält eine Liste mit den Namen der einzelnen Features, die verwendet werden.

Evaluation Zur Evaluierung der linearen Regression wurde eine ideale lineare Gerade erstellt. Mit Hilfe der Geraden wurden die einzelnen Werte bestimmt. Zudem wurde auf Basis des idealen Datensatzes, ein Datensatz mit Rauschen erstellt. Diese Werte wurden zum Evaluieren des Verfahrens verwendet. Mit Hilfe der Vorhersage und der gegebenen Daten konnte so der Mean-Squared-Error errechnet werden. Des Weiteren wurden die Größen Speichergröße, Datendurchsatz und Bearbeitungsdauer des Lerners betrachtet. In den nachfolgenden Diagrammen 69, 70 und 71 wird das Verfahren durch den roten Graphen repräsentiert und die ideale lineare Gerade durch den blauen Graphen.

Bei der Betrachtung der benötigten Speichergröße ist zu beobachten, dass das lineare Regresssionsverfahren mehr Speicher benötigt. Dies ist der Fall, da Daten zur Berechnung der Regressionsgeraden zwischengespeichert werden müssen, was bei der gegebenen Gerade nicht der Fall ist. Die lineare Regression benötigt ebenfalls nur konstant viel Speicher. Beim Datendurchsatz pro Zeit sind beide Verfahren gleich gut. Dies spricht für die lineare Regression, da genauso viele Beispiele bearbeitet werden. Zudem ist zu erkennen, dass das Verfahren nur wenig Zeit pro betrachtetem Beispiel benötigt. Die längste Bearbeitungszeit liegt hier bei ca. 1,65 ms. Die einzelnen

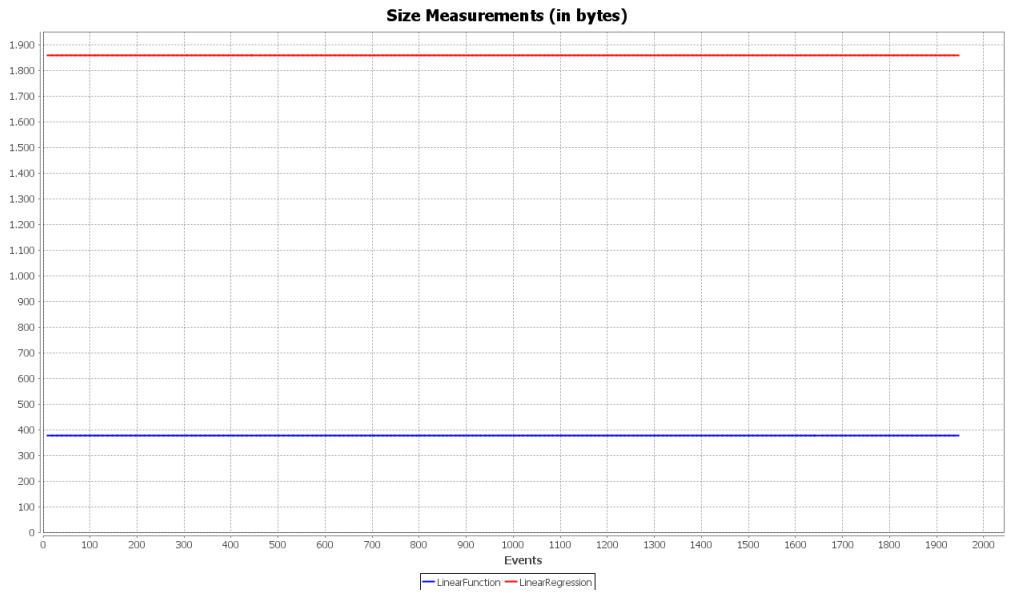


Abbildung 69: Verwendete Speichergröße

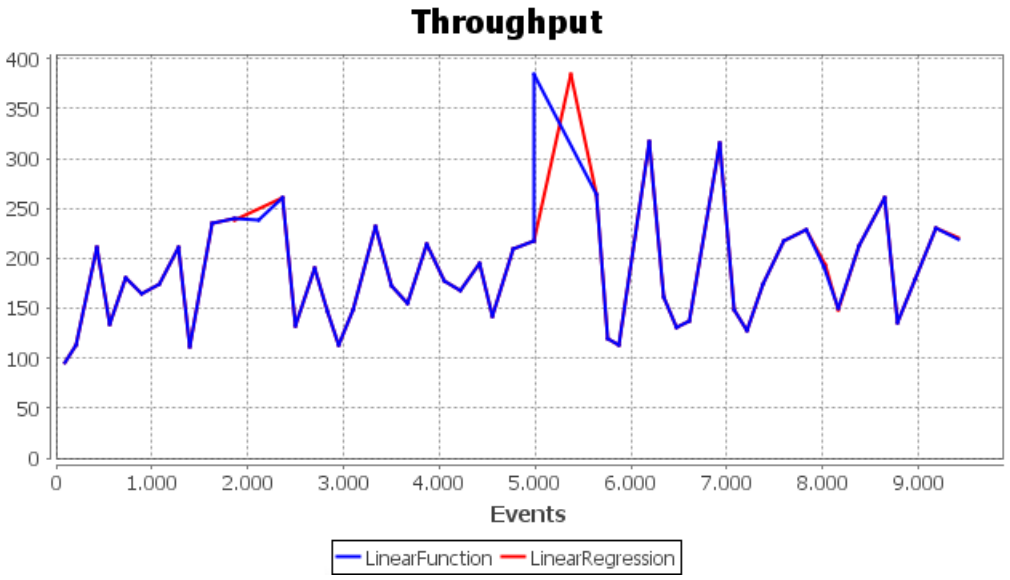


Abbildung 70: Datendurchsatz

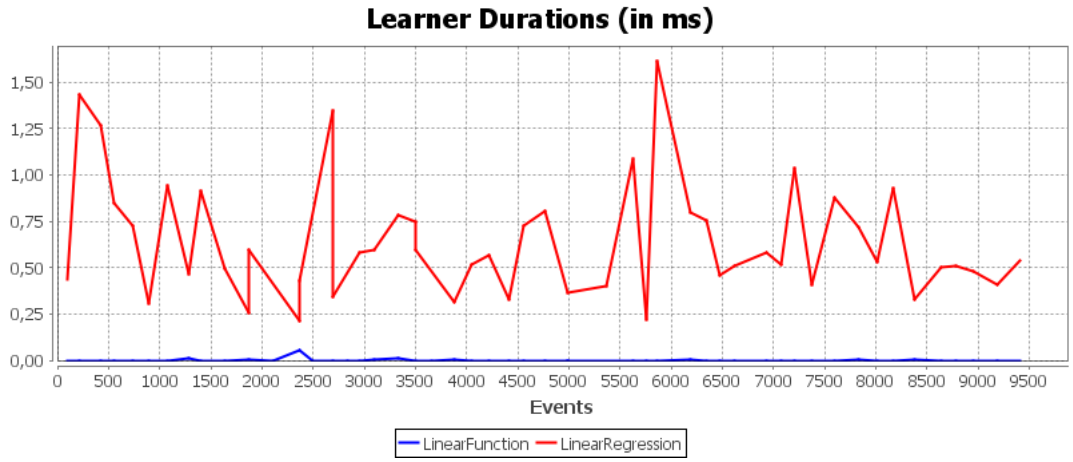


Abbildung 71: Bearbeitungsdauer für die einzelnen Beispiele

Ausschläge nach oben entstehen dadurch, dass in bestimmten Fällen, wie schon im Verfahren erläutert, die Regressionsgerade neu berechnet werden muss und somit der Learner mehr Zeit benötigt.

Die folgenden Diagramme 72 und 73 zeigen den Mean-Squared-Errors (MSE) für die lineare Regression. Dabei repräsentiert Abbildung 72 den idealen Datensatz und Abbildung 73 den Datensatz mit einem Rauschen. Das Rauschen entsteht durch einen Zufallswert zwischen -1 und 1, der auf den Featurewert aufaddiert wird. Bei der Betrachtung des MSE ist gut zu erkennen, dass es zu Beginn noch zu einer starken Abweichung kommt, da noch nicht die ideale Regressionsgerade bestimmt wurde. Im Laufe der Zeit wird der MSE immer besser, bis er schließlich gegen null geht. Dies ist der Fall, da mit der Zeit immer mehr Beispiele betrachtet wurden und so die Regressionsgerade immer besser der idealen Regressionsgeraden entspricht. Bei der Betrachtung des Datensatzes mit einem Rauschen verhält sich das Verfahren erwartungsgemäß und weist das selbe Verhalten wie der ideale Datensatz auf, dies ist in Abbildung 73 zu erkennen.

6.4.2 Regressionsbaum

Idee Ein Regressionsbaum verbindet die Eigenschaften von Regression und Entscheidungsbäumen in einem Algorithmus. Während bei Entscheidungsbäumen die Zielgröße einen nominalen Wert, also beispielsweise „wahr“ oder „falsch“ annimmt, ist die Zielgröße eines regressiven Verfahrens ein numerisches Attribut. Ziel des Algorithmus ist es, auf einer Menge von Beispielen mit numerischen und/oder nominalen Attributen einen Entscheidungsbaum zu lernen, in dessen Blättern regressive Abschätzungen der Zielgröße enthalten sind.

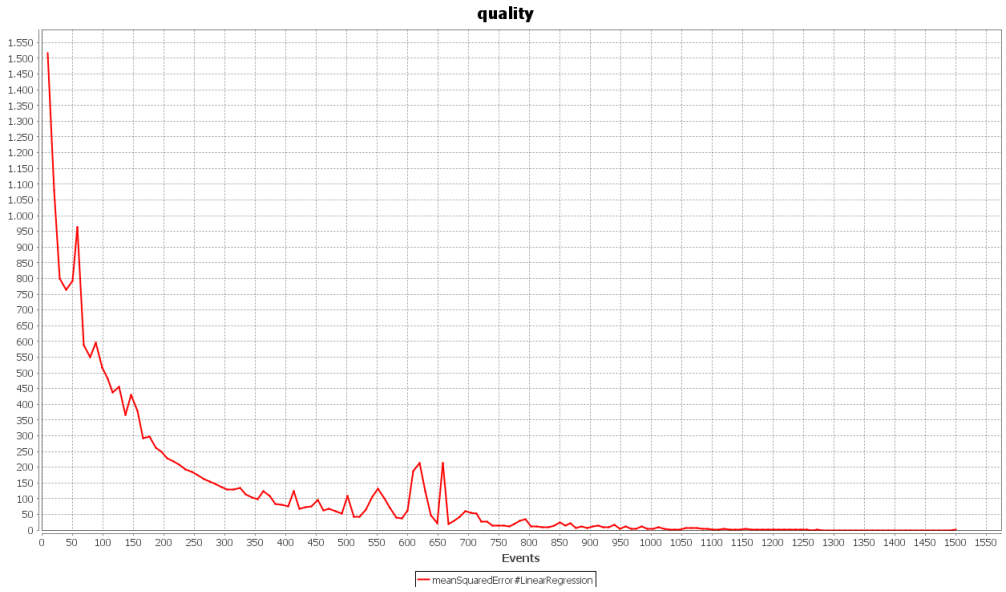


Abbildung 72: Vergleich des MSE

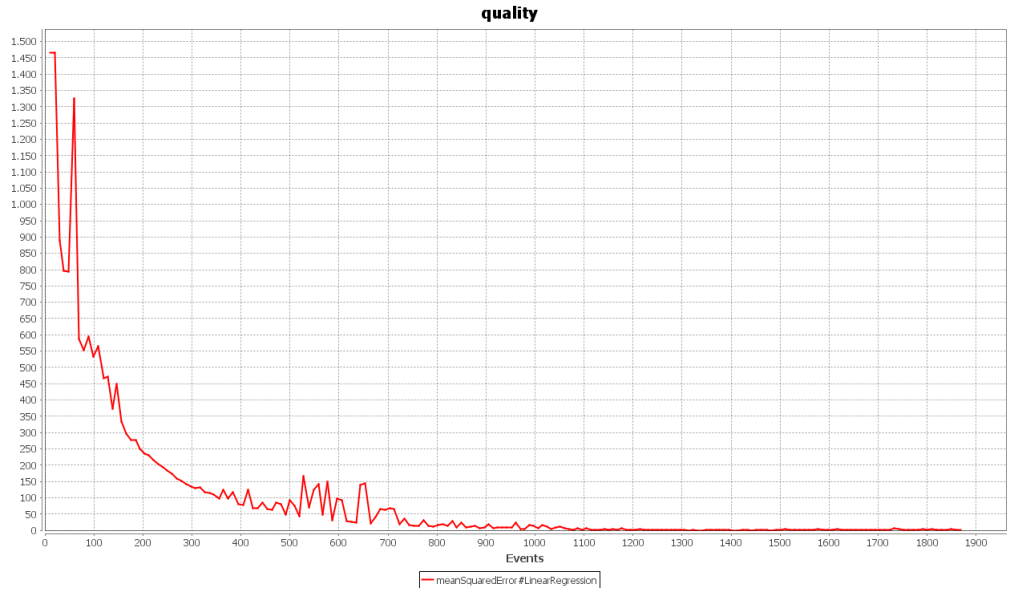


Abbildung 73: Vergleich des MSE bei Daten mit Rauschen

Algorithmus Die Implementierung basiert auf dem 2008 durch Gama und Ikonomovska in [28] vorgestellten FIMT-Algorithmus. Während die Blätter eines klassischen Regressionsbaum-Verfahrens direkt die Regressionswerte vorhalten, kommt hier der Ansatz eines Modell-Baums zum Einsatz. Die Blätter des gelernten Baums enthalten also keine numerischen Werte, sondern ein weiteres, lineares Regressionsverfahren. Dieses kann vom Nutzer frei gewählt und parametrisiert werden. In jedem

Blatt des Baums liegt somit zu jedem Zeitpunkt des Algorithmus ein valides lineares Regressionsmodell vor.

Die Hauptaufgabe des Algorithmus liegt somit in der Erzeugung des Entscheidungsbaums. In der Wurzel und in jedem inneren Knoten des Baums wird dazu eine Kleinergleich-Entscheidung (im Falle eines numerischen Attributs) bzw. eine Gleichheits-Entscheidung (bei nominalen Attributen) erzeugt. Anhand dieser Entscheidungen können eintreffende Beispiele vorklassifiziert werden, was zu einer deutlichen Verbesserung der Regressionsergebnisse führt.

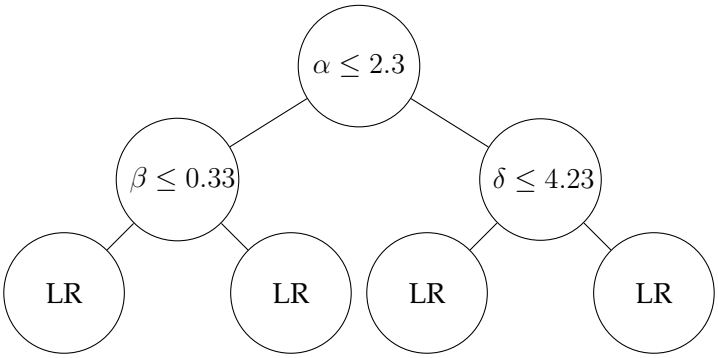


Abbildung 74: Regressionsbaum mit Entscheidungs- und Blattknoten. Attribute: α, β, δ, LR = lineares Regressionsmodell

Um Beispiele korrekt zu klassifizieren, muss die Wahl von Attribut und zugehörigem Wert optimal sein. Da keine Attribute von vornherein ausgeschlossen werden können, muss jeder mögliche *Splitpoint* betrachtet werden. Hierzu werden in jedem Blattknoten - neben dem linearen Regressionsmodell - auch noch statistische Informationen über alle Attribute vorgehalten und gesammelt. Als effiziente Datenstruktur hat sich hierbei ein binärer Baum je Attribut erwiesen. Anhand der gesammelten Statistiken lässt sich die Güte jeden möglichen Splitpoints errechnen. Diese wird aus dem Verhältnis zwischen der Standardabweichung vor und nach einem möglichen Split errechnet. Es wird also errechnet, welche Auswirkungen ein möglicher Split auf die Genauigkeit der Vorhersage hätte. Bezeichnet wird die Güte eines Splitpoints dabei als *Standard Deviation Reduction*.

Die Entscheidung, ob sich ein Attributwert als neuer Splitpoint eignet, wird mittels der Chernoff-Schranke getroffen. Anhand dieser kann die Wahl des richtigen Attributs mit Wahrscheinlichkeit $1 - \delta$ nachgewiesen werden, wobei δ ein vom Nutzer definiertes Fehlermaß ist.

Lernphase Sobald ein neues vorklassifiziertes Beispiel eintrifft, wird es anhand der Vergleiche in den inneren Baumknoten zu einem Blatt durchgereicht. Dort nimmt das Beispiel zum einen Einfluss auf das lineare Regressionsmodell, zum anderen dient es zur Aktualisierung der statistischen Daten innerhalb des Blatts. Da sich die Statistiken

der einzelnen Attribute verändert haben, könnte sich ein neuer möglicher Splitpoint ergeben haben. Daher wird der Splitpoint mit der größten Standard Deviation Reduction ermittelt und seine Nützlichkeit anhand der Chernoff-Schranke berechnet.

Erweist sich dieser Splitpoint als bestmögliche Wahl (erfüllt er die Vorgaben der Chernoff-Schranke), wird ein neuer Entscheidungsknoten an der Stelle des Blattknotens erzeugt und es entstehen zwei neue Blattknoten als Kinder dieses Knotens. Beide Kinder enthalten zu Beginn das lineare Regressionsmodell des ursprünglichen Blattknotens.

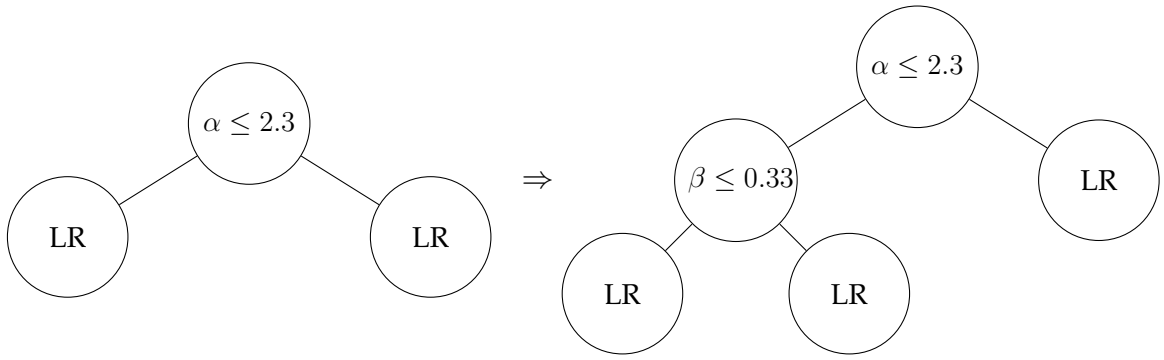


Abbildung 75: $\beta \leq 0.33$ wurde als neuer Splitpoint ausgewählt

Kann kein neuer Splitpoint gefunden werden, verändert sich die Baumstruktur nicht.

Vorhersagung Der Baum wird anhand der Entscheidungsknoten durchlaufen und das gegebene Beispiel dient als Eingabe für das lineare Regressionsmodell im erreichten Blatt. Die Vorhersage des linearen Regressionsmodells ist auch die Vorhersage und somit Ausgabe des Regressionsbaums.

Eingaben

Schlüssel	Typ	Bedeutung
event:learner:input	LabeledExample	Ein gelabeltes Example, auf dem gelernt werden kann

Ausgaben

Schlüssel	Typ	Bedeutung
event:learner:output	RegressionTree-Model	Modell zur Vorhersage des Regressionswerts

Parameter

Schlüssel	Typ	Opt.	Bedeutung
delta	double	Nein	durch den Benutzer definiertes Fehlermaß
linear-regression	String	Nein	URI des linearen Regressionsverfahrens, das in den Blättern des Baums genutzt werden soll

6.5 Clustering

Clustering bezeichnet Analyseverfahren, die eine Menge von Objekten in Teilmengen mit ähnlichen Merkmalen splitten. Da Clustering-Verfahren bietet vielfältige Einsatzmöglichkeiten beispielsweise bei der automatischen Klassifizierung, der Bildverarbeitung, der Mustererkennung sowie dem Text Mining. Die Projektgruppe hat sich daher entschieden, zwei unterschiedliche Clustering-Verfahren für Streams zu implementieren. Die Wahl ist hierbei auf zwei der bekanntesten und effizientesten Algorithmen gefallen: *BIRCH* [42] und *D-Stream* [12].

6.5.1 BIRCH

Idee Obwohl die Veröffentlichung von BIRCH etwas weiter zurückliegt, zählt er immer noch zu den effizientesten Clustering-Algorithmen auf Streams, so dass die Leistung von neuveröffentlichten Algorithmen für diese Lernaufgabe stets mit BIRCH verglichen werden.

Da es in der Welt des Clusterings viele verschiedene Distanzmaße gibt, die für bestimmte Datensätze sinnvoll sind, speichert BIRCH für jedes Cluster ein sogenanntes *ClusteringFeature (CF)*, welches alle notwendige Informationen enthält um Distanzen mittels verschiedener Distanzmaße zu berechnen. Ein *ClusteringFeature (CF)* besteht dabei aus drei Werten: N, die Anzahl der Datenpunkte in dem Cluster, $\vec{LS}$, die Lineare Summe der Datenpunkte in dem Cluster, und $\vec{SS}$ die quadratische Summe der Daten. Laut dem Additivitäts-Theorem können nun, falls zwei Cluster zusammengefasst werden, die Werte in dem ClusteringFeature der beiden Cluster einfach aufaddiert werden, um das entstehende Cluster zu beschreiben.
Die, durch die *ClusteringFeatures* beschriebenen, Cluster, werden nun in einem Baum gespeichert, dem *CFTree*. Dieser besteht aus inneren Knoten, die B Kinder haben können und für jedes Kind das ClusteringFeature des Kindclusters speichern. Die Blätter speichern nur L ClusteringFeatures. Die Parameter B und L hängen von der sogenannten page-size P ab.

Algorithmus Der Algorithmus besteht im Prinzip aus zwei Phasen: Der Insert- und der Rebuilding-Phase. In der Insert-Phase werden solange Datenpunkte in den *CFTree* eingefügt, bis der Hauptspeicher des Rechners voll ist. Dann beginnt das Rebuilding, in der die Parameter des Algorithmus verändert und aus dem alten *CFTree* ein neuer *CFTree* generiert wird, der weniger Hauptspeicher braucht.

Insert-Phase

Die Eingabe für die Insert-Phase ist ein mehrdimensionaler Datenpunkt. BIRCH durchläuft nun den bisherigen *CFTree* von oben nach unten bis er in einem Blatt landen. Dabei wählt er in jedem Knoten das Kind, dessen *ClusteringFeature* nach dem, vom

Nutzer gewähltem, Distanzmaß am Nächsten ist.
Hat der Algorithmus nun ein Blatt erreicht, sucht er aus den *ClusteringFeatures* des Blattes das Naheste aus. Jetzt wird getestet, ob dieses *ClusteringFeature* noch einen weiteren Datenpunkt aufnehmen kann. Wenn ja, wird dieser aufgenommen. Wenn nein, wird im Blattknoten geprüft, ob dort noch Platz für einen weiteren *ClusteringFeature*-Eintrag ist. Wenn dies möglich ist, erzeugt BIRCH diesen mit dem Datenpunkt als einzigen Punkt. Ansonsten muss der Algorithmus den Knoten *splitten*.
Beim *Split* werden die entferntesten *ClusteringFeatures* als Startpunkte für die beiden neuen Knoten gewählt und die restlichen *ClusteringFeature* auf den jeweiligen nahesten Knoten verteilt.
In jedem Fall müssen nach dem Einfügen des Datenpunktes die *ClusteringFeatures* des Pfades vom Blatt bis zur Wurzel aktualisiert werden. Falls kein *Split* erfolgte, heißt dies einfach, dass der Datenpunkt auf die *ClusteringFeatures* der auf dem Pfad liegenden Knoten addiert wird. Im Falle eines *Splits* muss ein neues Kind im Elternknoten eingetragen werden. Falls im Elternknoten kein Platz für ein weiteres Kind ist, muss dieser auch *gesplittet* werden. Das Ganze kann sich rekursiv bis zur Wurzel ziehen, dann erhöht sich die Tiefe des Baumes um eins.
Im letzten Schritt werden im Gegensatz dazu wieder Knoten *gemergt*: Falls es zu einem *Split* kam, gibt es einen Knoten an dem der *Split* nicht mehr weitergereicht wurde. Diesen Knoten scannt BIRCH nun, nimmt die beiden nahesten Einträge und falls dies nicht die Knoten sind, die für den *Split* verantwortlich waren, *mergt* er diese. Falls der entstehende Knoten nun mehr Einträge besitzt als erlaubt, wird dieser wieder *gesplittet*.

Rebuilding-Phase

In der Rebuilding-Phase werden alle Pfade von der Wurzel bis zu den Blättern des *CFTrees* durchlaufen. Der aktuelle Pfad nennt sich *OldCurrentPath* und ist am Anfang der linkeste Pfad des Baumes. Während des Rebuilding-Prozesses wird ein neuer *CFTree* generiert, dessen Wurzel mit null initialisiert wird. Darüber hinaus wird der Parameter T erhöht.
Für jeden *OldCurrentPath* passiert jetzt Folgendes: Als Erstes wird der zu *OldCurrentPath* korrespondierende Pfad im neuen *CFTree* eingefügt. Dieser wird *NewCurrentPath* genannt. Jetzt ermittelt BIRCH allerdings für die Blatteinträge des *OldCurrentPaths* den Pfad zu den Blättern mittels dem Distanzmaß neu. Den dabei entstehenden Pfad nennt sich *NewClosestPath*. Passen die Blatteinträge jetzt in das Blatt des *NewClosestPath* und ist *NewClosestPath before NewCurrentPath* fügt man sie dort ein. *Before* bedeutet hier, dass, wenn alle Blätter des Baumes mit Tiefe h mit i durchnummeriert sind, Pfad A mit $(i_1^1, \ldots, i_h^1)$ *before* Pfad B mit $(i_1^2, \ldots, i_h^2)$ ist, falls $i_1^1 \leq i_1^2, \ldots, i_h^1 \leq i_h^2$ gilt. Ansonsten werden die Blatteinträge in *NewCurrentPath* eingefügt.
Sobald alle Blatteinträge von *OldCurrentPath* betrachtet wurden, können alle nicht weiter verwendeten Knoten auf diesem Pfad gelöscht werden. Daraufhin wird *OldCurrentPath* auf den nächsten Pfad des alten *CFTrees* gesetzt.

Umsetzung Die Klassen des Algorithmus BIRCH sind auf mehrere Pakete des Projekts *pg542-util* verteilt:

- *edu.udo.cs.pg542.util.clustering.birch*
  - *BIRCH:*
    Klasse des globalen Algorithmus: Verarbeitet ankommende *Examples*, *inserted* diese in den aktuellen CFTree ein oder *rebuildet* diesen, falls die Anzahl der gesehenen *Examples* den Rebuild Counter überschreitet.
- *edu.udo.cs.pg542.util.data.model*
  - *CFTree:*
    Repräsentierende Klasse des *CFTrees*. Hier sind die beiden Methoden *insert()* und *rebuild()* definiert. Darüber hinaus werden hier die Parameter festgelegt und bestimmt.
  - *CFTreeNode:*
    Repräsentiert sowohl innere Knoten als auch Blattknoten des Baumes. Hier wurden Methoden implementiert, die für die Verarbeitung der Datenpunkte in einzelnen Knoten wichtig sind.
  - *ClusteringFeature:*
    Realisiert das ClusteringFeature.
- *edu.udo.cs.pg542.util.math.distance*
  Hier finden sich mehrere Klassen, die jeweils ein bestimmtes Distanzmaß realisieren, welche u.a. von BIRCH genutzt werden können. In der aktuellen Implementierung verwendet BIRCH die euklidische Distanz zur Abstandsberechnung.

Knotenklasse

edu.udo.cs.pg542.util.node.learner.BirchNode

Eingaben

Schlüssel	Typ	Bedeutung
event:learner:input	Example	Mehrdimensionale numerische Datenpunkte

Ausgaben

Schlüssel	Typ	Bedeutung
event:learner:output	Integer	Wert, welcher das vorhergesagte Cluster bezeichnet

Parameter

Schlüssel	Typ	Opt.	Bedeutung
T	Double	Ja	Threshold T: Gibt die Toleranz der Distanz innerhalb der Punkte eines Clusters an. Wird von BIRCH automatisch erhöht.
P	Double	Ja	Page-Size P: Gibt die maximale Größe an Datenpunkten an, die ein Cluster enthalten kann. Hängt von der Dimension der Daten ab.
REBUILDCOUNTER	Integer	Ja	Rebuild Counter: Gibt die Anzahl der Events an, nach denen der CFTree rebuilded wird. Sollte so gewählt werden, dass kurz bevor der Hauptspeicher voll ist, rebuilded wird.
PREDICTIONLEVEL	Integer	Ja	Predicition Level: Gibt die Ebene des *CFTrees* an, auf welcher die Prediction stattfindet.

Evaluation Der Algorithmus BIRCH wurde mit zwei verschiedenen Datensätzen evaluiert, zum einem mit dem recht bekannten Iris-Datensatz, zum anderem mittels generierten Daten. Der Iris-Datensatz besteht aus 4 numerischen Attributen und einem Label, dass 3 verschiedene Cluster beschreibt. Der generierte Datensatz dagegen beinhaltet 3 numerische Attribute mit einem Wertebereich von 0 bis 1 und einem Label, das den Datenpunkten eines von 7 Clustern zuweist.

Iris-Daten

Der Iris-Datensatz wurde mit dem Default-Threshold T von 0.0, einer page-size P von 12.0, einem Rebuild Counter-Wert von 10000 und einem Prediction Level von 1 evaluiert.

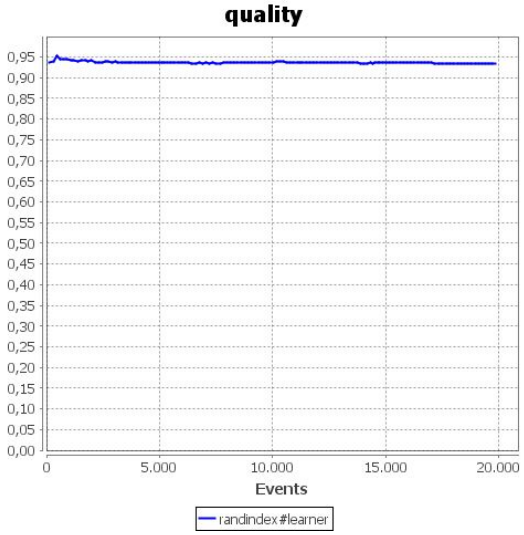


Abbildung 76: RandIndex auf Iris-Daten

Die Qualität von BIRCH wurde mittels des RandIndexes berechnet. Wie in Abbildung 76 zu sehen hält sich dieser nahezu konstant bei 0.94 und verändert sich auch nicht nach dem Rebuild beim 10.000 Event.

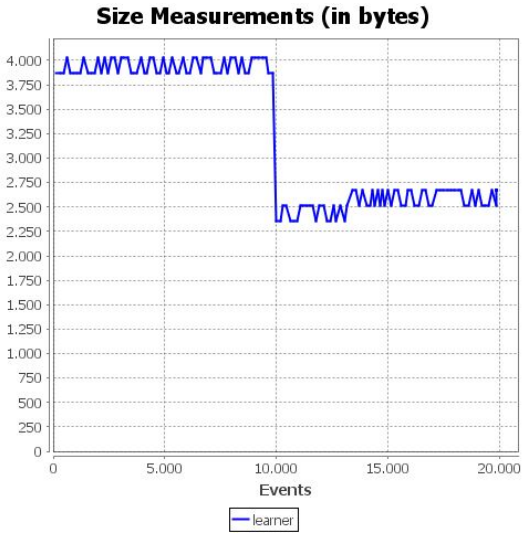


Abbildung 77: Messung der Größe des Modells auf Iris-Daten

Die Größe des Modells (Abb. 77) hält sich direkt nach dem ersten Event, bis auf kleine Schwankungen, konstant bei ca. 4.000 Byte. Schön zu sehen ist hier, wie der Rebuild-Prozess beim 10.000 Event die Größe des Modells auf ca. 2.500 Byte reduziert. Auch danach steigt die Größe des Modells nur leicht an.

Die Lernzeit ist, wie in Abbildung 78 zu sehen, bei den ersten ankommenden Events mit ca. 0,65 am höchsten und nimmt danach kontinuierlich ab. Dabei unterliegt die

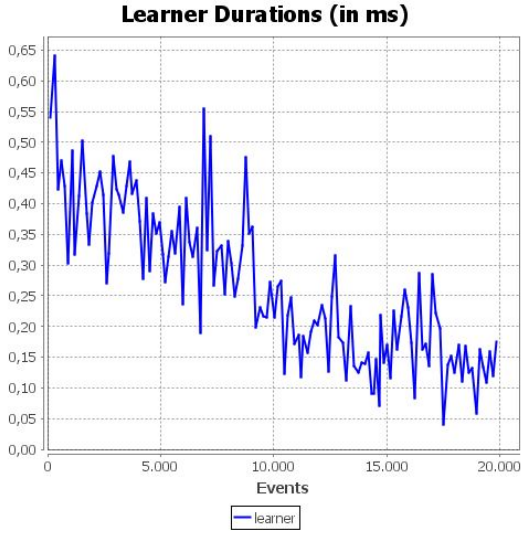


Abbildung 78: Messung der Lernzeit auf Iris-Daten

Zeit zum Lernen aber während des ganzen Prozesses Schwankungen, der Rebuild-Vorgang beim 10.000 Example macht sich allerdings nicht bemerkbar.

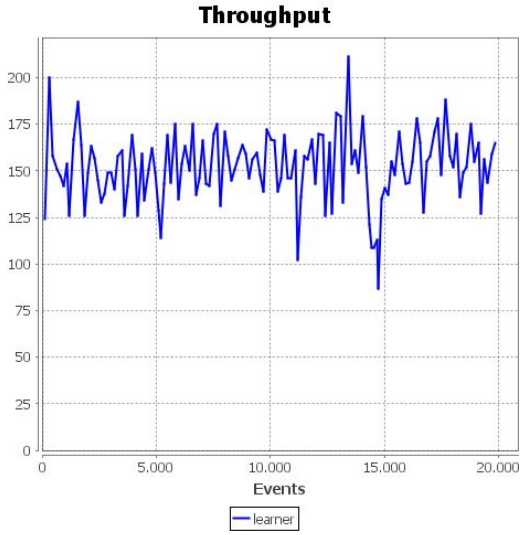


Abbildung 79: Messung der Durchsatzes auf Iris-Daten

Der Durchsatz (Abb. 79) unterliegt während der Verarbeitung aller 20.000 Events recht starken Schwankungen, schwingt allerdings insgesamt recht konstant um 150.

Generierte Daten

Der generierte Datensatz wurde mit dem Default-Threshold T von 0.0, einer page-size P von 256.0, einem Rebuild Counter-Wert von 10000 und einem Predicition Level von 2 evaluiert.

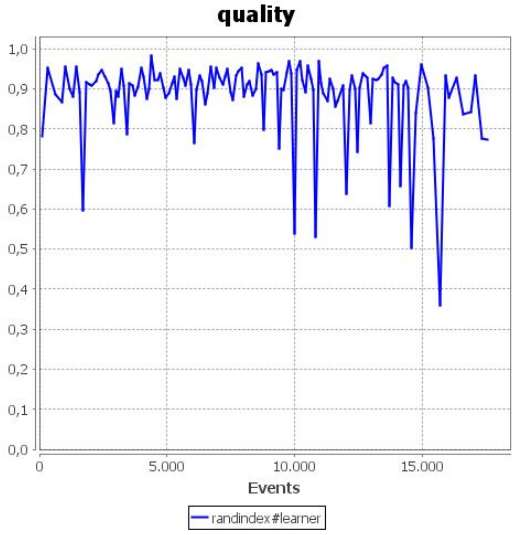


Abbildung 80: RandIndex auf generierten Daten

Im Gegensatz zum RandIndex auf dem Iris-Datensatz schwankt die Qualität der Vorhersage auf den generierten Daten sehr stark (siehe Abb. 81). Im Prinzip hält sich diese bei 0.9, es gibt aber, besonders nach dem Rebuild-Prozess bei Event 10.000, Ausreißer-Werte nach unten.

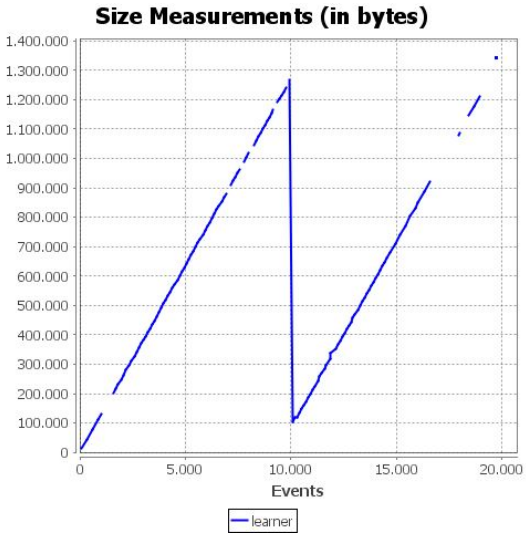


Abbildung 81: Messung der Größe des Modells auf generierten Daten

Auch hier (Abb. 81) sieht man, dass die Größe des Modells stark von den Daten abhängt: Anders als bei den Iris-Daten steigt die Größe des Modells linear bis zum Rebuild an. Dieser reduziert die Größe des Modells stark auf ca. 100.000 Byte, danach steigt diese wieder linear an. Die Messung der Größe des Modells war anscheinend nicht während des ganzen Prozesses möglich, trotzdem ist der Verlauf deutlich zu erkennen.

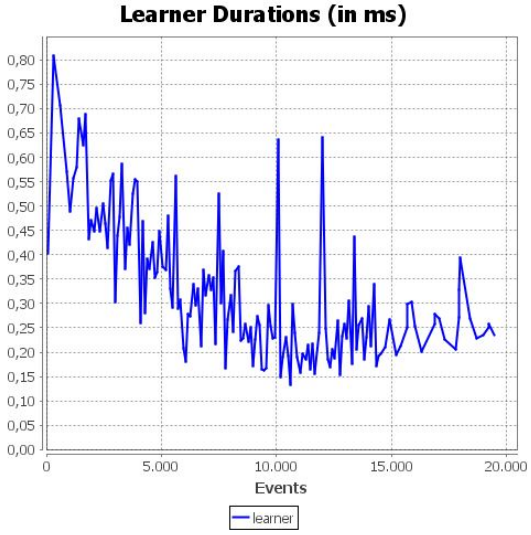


Abbildung 82: Messung der Lernzeit auf generierten Daten

Die Lernzeit ist hier (Abb. 82), wie auch auf den Iris-Daten, bei den ersten ankommenden Events am Größten, sinkt dann und hält sich danach ungefähr konstant bei 0.25 ms. Dabei unterliegt die Lernzeit auch hier starken Schwankungen, besonders kurz nach dem Rebuild reißt diese nach oben aus. Ob dies unmittelbar mit dem Rebuild zusammenhängt ist schwer zu sagen, da dies nach ca. 12.000 Events erneut passiert.

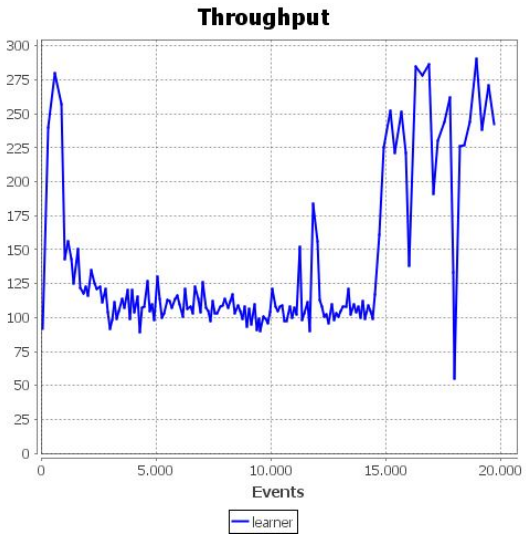


Abbildung 83: Messung der Durchsatzes auf generierten Daten

Der Durchsatz ist, wie in Abbildung 79 zu sehen, auf den generierten Daten am Anfang recht hoch, liegt dann aber bis zum 15.000 Event recht konstant bei ca. 110. Danach steigt der Durchsatz allerdings wieder stark an und hält sich nun mit Ausreißern nach unten bei ca. 250.

6.5.2 D-Stream

Idee Der Clustering-Algorithmus *D-Stream* wurde im Jahr 2007 von Yixin Chen und Li Tu [12] vorgestellt. Die Autoren wollten mit diesem Algorithmus den Schwächen des *CluStream* - Algorithmus begegnen und ein neues Verfahren präsentieren, dass diese Schwächen nicht aufweist.

D-Stream arbeitet - wie CluStream auch - in zwei Phasen

1. einer Online-Phase, die das micro-Clustering durchführt und
2. einer Offline-Phase, die auf der Basis der micro-Cluster ein endgültiges macro-Clustering durchführt.

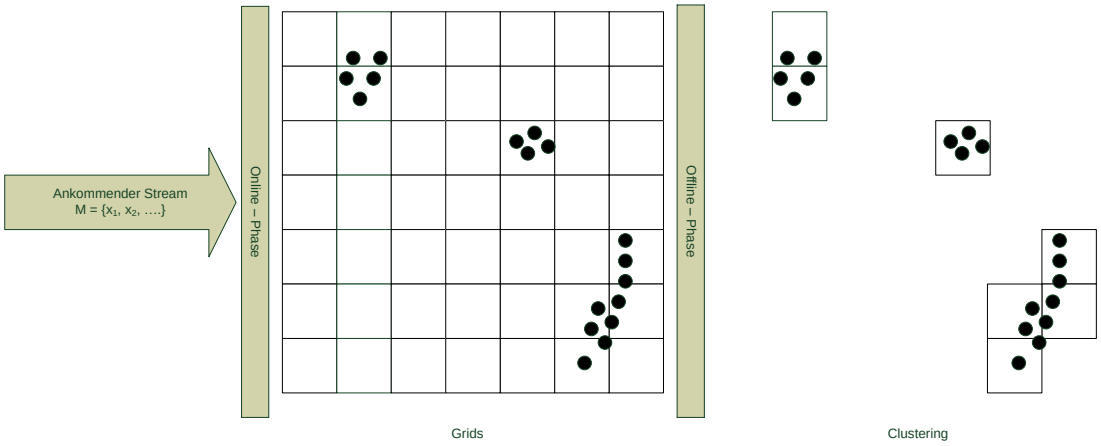


Abbildung 84: Funktionsweise von D-Stream

Voraussetzung um D-Stream anzuwenden ist, dass alle ankommenden Datenpunkte in dem n-dimensionalen Datenraum S definiert sind.

$$S = S_1 \times S_2 \times S_3 \times ... \times S_n$$

Zunächst partitioniert D-Stream S in eine endliche Menge von *desity grids*. Hierzu wird jede Dimension in p_i disjunkte Teilmengen aufgeteilt. Die Teilmengen sollten gleich große Wertebereiche umfassen.

$$S_i = S_{i,1} \bigcup S_{i,2} \bigcup ... \bigcup S_{i,p_i}$$

Jeder ankommende Datenpunkt kann nun eindeutig auf ein *density grid* gemappt werden. Für jedes *density grid*, das eine Dichte über einem gewissen Schwellwert hat, wird nun ein charakteristischer Vektor gespeichert und ein Eintrag in der $gridListe$

erzeugt. Der charakteristische Vektor eines Grids [7] g besteht dabei auf dem Fünftupel $(t_g, t_m, D, label, status)$, wobei

- t_g die Zeit angibt, an der der Vektor zuletzt aktualisiert wurde,
- t_m die Zeit angibt, an der der Vektor zuletzt aus der $gridListe$ entfernt wurde,
- D die Dichte des Grids angibt,
- $label$ die Nummer des Clusters angibt, zu dem das Grid gehört und
- $status = \{SPORADISCH, NORMAL\}$ ein Label zum Entfernen sporadischer Grids (= Grids, deren Dichte unter den Schwellwert gefallen ist) aus der $gridListe$ ist.

Anhand ihrer Dichte D unterscheiden wir zu jedem Zeitpunkt t drei Arten von Grids:

1. dichte Grids (*dense grids*)

$$D(g,t) \geq \frac{C_m}{N(1-\lambda)} = D_m$$

   C_m mit $C_m > 1$ ist dabei ein Parameter, der den Grenzwert für dichte Grids reguliert.

2. Übergangs-Grids (*transitional grids*)

$$\frac{C_l}{N(1-\lambda)} \leq D(g,t) \leq \frac{C_m}{N(1-\lambda)}$$

   C_l mit $1 > C_l > 0$ ist dabei ein Parameter, der den Grenzwert für spärliche Grids reguliert.

3. spärliche Grids (*sparse grids*)

$$D(g,t) \leq \frac{C_l}{N(1-\lambda)} = D_l$$

Da die Dichte jedes Grids über die Zeit durch Anwendung einer Verfallsfunktion abnimmt, ist die Anzahl dichter Grids nach oben beschränkt. Die Verfallsfunktion wird natürlich nur angewendet, wenn das Grid entweder zur Berechnung des Modells oder aufgrund des Eintreffen eines neues Datenpunktes, der auf der Grid gemappt wurde, aktualisiert werden muss. So wird beim Eintreten eines dieser Ereignisse zum Zeitpunkt t_c die Dichte des Grids auf

$$D(g,t_c) = D(g,t_g) \times \lambda^{t_c - t_g} + 1$$

[7] Der englische Begriff grid wird im Folgende äquivalent zu den deutschen Wörtern „Planquadrat“ und „Rasterfeld“ verwendet.

gesetzt. Anhand der Griddichte wird dann in der Offline-Phase das Modell gebildet. Hierzu werden zunächst alle dichten Grids als eigene Cluster gelabelt. Dann werden zusammenhängende Cluster zusammengeführt. Zwei Cluster werden genau dann als zusammenhängend bezeichnet, wenn sie durch eine Menge dichter Grids oder Übergangs-Grids verbunden sind.

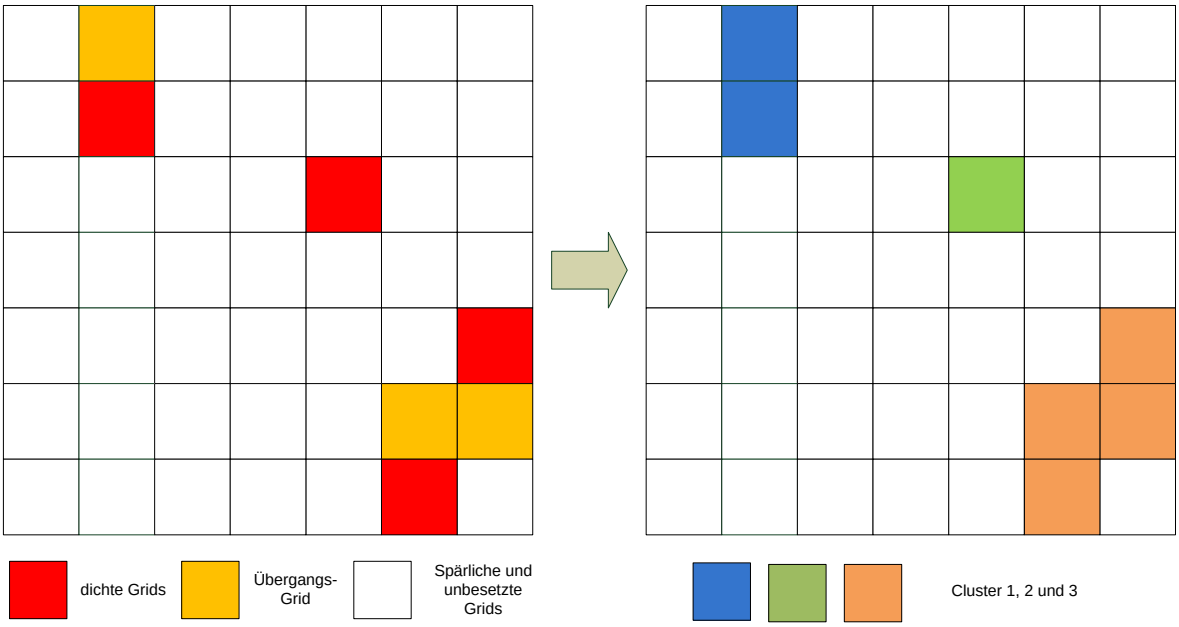


Abbildung 85: Macro-Clustering: Die Erstellung von Clustern auf Basis der Griddichte

Umsetzung Um die Implementierung des Verfahrens zu vereinfachen, wird angenommen, dass die ankommenden Datenpunkte für jede Dimension einen Wert zwischen 0 und 1 aufweisen.

$$0 \leq S_i \leq 1, \forall S_i$$

Außerdem wird darauf verzichtet, die Anzahl der *grids* für jede Dimension individuell festzulegen. Die „Auflösung“ kann lediglich für alle Dimension durch Ändern des *resolution*-Parameters optional geändert werden. Darüber hinaus werden Examples, die zu wenig Attribute aufweisen, oder bei den die Attribute nicht zu den erwarteten Attributen passen, komplett verworfen. Welche Attribute von einem Example erwartet werden, wird anhand des ersten ankommenden Examples festgelegt. Es wird davon ausgegangen, dass etwaige Veränderungen an den zu beachtenden Examples in Form einer preprocessing-Komponente vorgenommen werden.

Knotenklasse

edu.udo.cs.pg542.util.node.learner.DStreamNode

Eingaben

Schlüssel	Typ	Bedeutung
event:learner:input	Example	Mehrdimensionaler, numerischer, auf einen Wertebereich zwischen 0 und 1 genormter Datenpunkte

Ausgaben

Schlüssel	Typ	Bedeutung
event:learner:output	DStreamModel	Das gebildete Modell

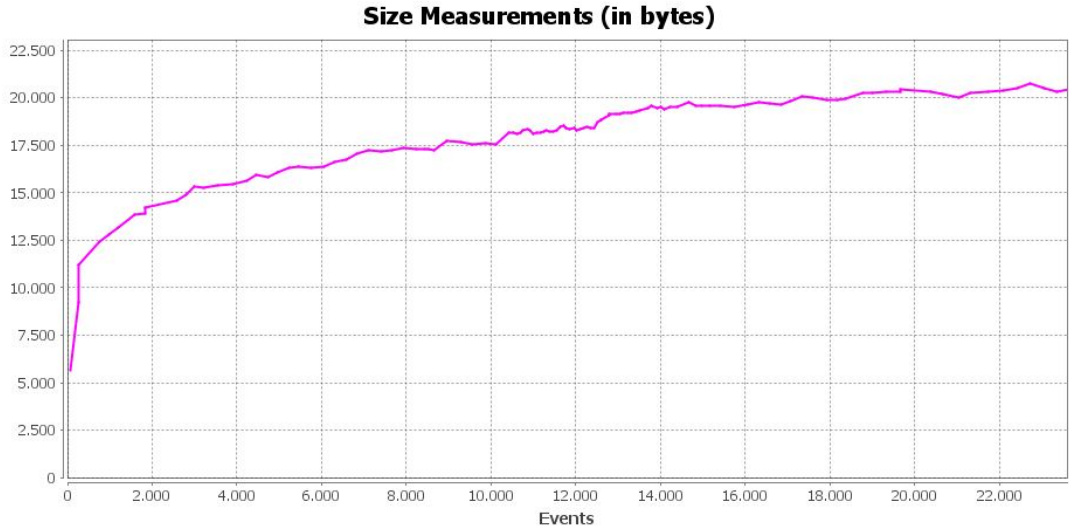
Parameter

Schlüssel	Typ	Opt.	Bedeutung
features	List<String>	Nein	Liste der Featurekeys, die von dem Lerner für die Modellbildung beachtet werden sollen
resolution	Integer	Ja	Gibt an, in wie viele Grids jede Dimension unterteilt werden soll
cm	Double	Ja	Reguliert den Grenzwert für dichte Grids
cl	Double	Ja	Reguliert den Grenzwert für spärliche Grids
decay	Double	Ja	Verfallsfaktor für den Dichte-Verfall

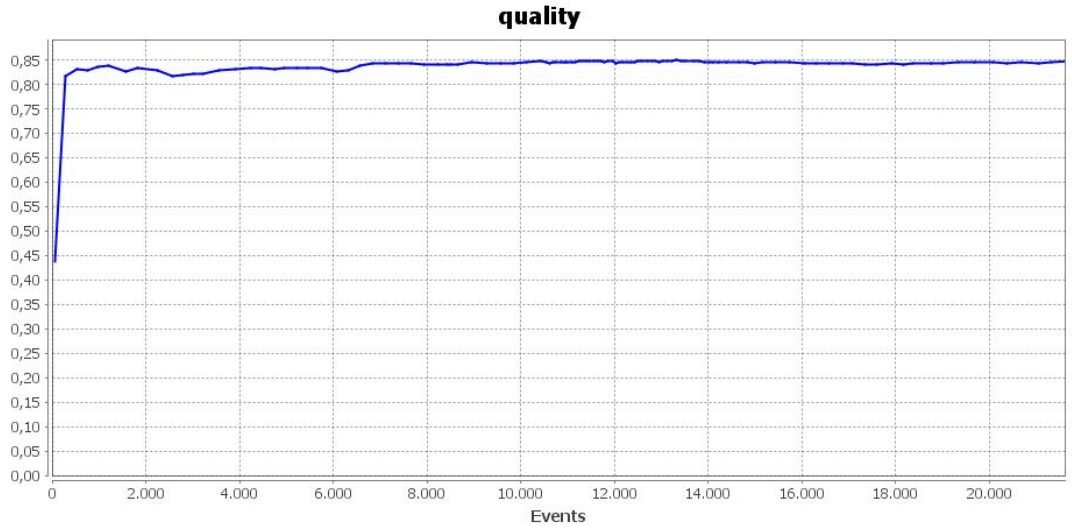
Evaluation Die Evaluierung des DStream-Verfahrens wurde nur mit einem generierter Datensatz durchgeführt. Als Qualitätskriterium wurde der Rand-Index zwischen den erzeugten Labeln und den Labeln der generierten Daten verwendet.

Es zeigt sich, dass die Qualität der Ergebnisse sehr stark von einer geschickten Wahl der Lerner-Parameter abhängt. Im Folgenden finden sich die Evaluationskurven des Verfahrens mit den Parameters resolution=20, cm=3.0, cl=0.3 und decay=0.997. Auffällig ist, das schon nach einer sehr geringen Anzahl gesehener Beispiel der Rand-

Index einen relativ hohen Wert annimmt, der dann im Folgenden annährend konstant bleibt. Dieser Sachcerhalt kann in Abbildung 86b nachvollzogen werden. Der Speicherverbrauch (Abbilung 86a) des Algorithmus nährt sich einer oberen Schranke an, ist aber tendenziell steigend, da die Anzahl der belegten Grids, für die Informationen vorgehalten werden, natürlich stetig ansteigt. Da der Algorithmus Informationen über sehr spärlich besetzte Grids nach einer gewissen Zeit aber auch verwerfen kann, erreicht der gesamte Speicherverbrauch nie den Maximalwert, den eine Speicherung aller Informationen über alle Grids erfordern würde.



(a) Speicherverbrauch des D-Stream Verfahrens



(b) Qualität des DStream-Verfahrens (Rand-Index)

Abbildung 86: Evaluierung von D-Stream

6.6 Verteiltes Lernen

Das Überwachen von Schwellwertverletzungen in verteilten Stream-Szenarien ist eine anspruchsvolle Aufgabe, die üblicherweise viel Kommunikation bedarf. Sharfman et al. stellten 2006 einen geometrischen Ansatz vor, um beliebige Schwellwertfunktionen über verteilten Streams zu überwachen und dabei die Kommunikation zu minimieren [37]. Auf diesem Ansatz basiert das hier implementierte Verfahren ***G**eometric **T**hreshold **A**pproach (GTA)*.

Szenario
Es gebe n lokale Einheiten, die unabhängig voneinander, verteilt Daten über verschiedenen Streams erheben. Die lokal erhobenen Daten müssen gleichförmig sein, d.h. wenn man den Mittelwert aus allen lokalen Daten berechnet, erhält man dasselbe Ergebnis, dass ein Rechner erhielte, der sämtliche Streams gleichzeitig verarbeiten könnte und aus allen gesehenen Elementen mit demselben Verfahren die Daten erheben würde.
Die lokal erhobenen Daten werden *lokale Statistiken* oder *lokale Statistikvektoren* genannt, der Mittelwertvektor aller n lokalen Statistikvektoren wird als *globale Statistik* oder *globaler Statistikvektor* bezeichnet. Alle Statistikvektoren müssen $\in R^m$ sein.
Zusätzlich ist eine Schwellwertfunktion $f : R^m \longrightarrow \{true, false\}$ gegeben, die für jedes $\vec{x} \in R^m$ definiert, ob der Schwellwert überschritten ist.
Die Aufgabe besteht darin, eine Schwellwertüberschreitung der globalen Statistik zu detektieren. Lokale Schwellwertüberschreitungen werden dabei nicht überwacht.

Um diese Aufgabe zu lösen, würde ein naiver Algorithmus jede Änderung einer lokalen Statistik zu einer zentralen Einheit senden, die zu jedem Zeitpunkt alle aktuellen lokalen Statistiken kennt und somit die globale Statistik berechnen und überwachen kann.
Für das hier angenommene Szenario ist jedoch Kommunikation teuer und soll daher minimiert werden.

Die wesentliche Idee des Verfahrens basiert darauf, dass sich der globale Statistikvektor innerhalb der konvexen Hülle befindet, die von allen lokalen Statistikvektoren aufgespannt wird, und dass die konvexe Hülle von der Fläche aller Kreise (2D zur einfacheren Vorstellung, Verfahren gilt für beliebige Dimensionalität) abgedeckt wird, auf deren Rand jeweils der globale und ein lokaler Statistikvektor liegen.
Jede lokale Einheit überwacht nun ihren *lokalen Kreis*. Überschreitet kein Punkt des Kreises den Schwellwert, ist lokal sichergestellt, dass der globale Statistikvektor den Schwellwert ebenfalls nicht verletzt. Überschreitet ein Punkt des Kreises den Schwellwert, müssen sich die Einheiten synchronisieren: Die lokalen Einheiten senden ihre lokalen Statistiken zu einer zentralen Einheit. Die zentrale Einheit berechnet dann die globale Statistik und sendet diese wieder an die lokalen Einheiten. Bei jeder Synchronisation kann die zentrale Einheit eine tatsächliche Verletzung des Schwellwertes der globalen Statistik feststellen.

Nach einer Synchronisation erhalten alle lokalen Einheiten die neu berechnete globale Statistik, die sie als *estimate Vektor* speichern. Zudem speichern sie den bei der Synchronisation vorhandenen lokalen Statistikvektor. Mit diesem und dem jeweils aktuellen lokalen Statistikvektor können sie den *lokalen Driftvektor* als Differenz der beiden Vektoren berechnen.
Wird nun die lokale Statistik aktualisiert, untersucht jede lokale Einheit den Kreis auf dem der estimate Vektor und der estimate Vektor + Driftvektor liegen.

Implementierungsdetails
Alle GTA-Klassen befinden sich im Paket *edu.udo.cs.pg542.util.monitoring*.
Damit ein Knoten eine lokale GTA Einheit repräsentiert, muss der Knoten statt von *AbstractNode* von *GTALocalUnit* erben. Hierdurch erhält er zwei Parameter: SAMPLING_AMOUNT (Anzahl der zu testenden Punkte auf dem Kreis) und THRESHOLD (die Schwellwertfunktion). Die Schnittstelle zum GTA-Subsystem bildet die Methode *updateLocalStatistics(List)*, die einen als *List<Double>* repräsentierten lokalen Statistikvektor erhält. Mit dieser Methode können die lokalen Statistiken aktualisiert werden. Die Methode hat jedoch keinen Effekt, falls die lokale Einheit gerade auf Antwort von der zentralen Einheit wartet. Mittels *isWaiting()* kann geprüft werden, ob die lokale Statistik aktualisiert werden kann.

Kommunikation
Die Kommunikation erfolgt ausschließlich über Signale an Topics. Ein explizites Verknüpfen von Knoten ist somit nicht erforderlich.

Topics (zu finden in *GTACentralUnit*)

- GTA_LOCAL_NODES_TOPIC um die lokalen Einheiten zu erreichen
- GTA_CENTRAL_NODE_TOPIC um die zentrale Einheit zu erreichen
- GTA_GLOBAL_VIOLATION_TOPIC Zu diesem Topic müssen sich alle Knoten anmelden, die benachrichtigt werden und reagieren wollen, falls die globale Statistik den Schwellwert überschreitet.

Das GTA-Subsystem kann zurückgesetzt werden (z.B. nach einer globalen Schwellwertüberschreitung), indem *GTA_RESET* an die zentrale Einheit gesendet wird.

Parameter

Schlüssel	Typ	Opt.	Bedeutung
sampling-amount	Integer-ParameterType	Nein	Für die Prüfung ob ein Punkt des Kreises den Schwellwert überschreitet, werden *SAMPLING_AMOUNT* viele zufällige Punkte auf dem Kreis getestet. Wenn der Wert 0 ist, werden deterministisch immer alle Schnittpunkte des Kreises mit allen Koordinatenachsen getestet, die existieren würden, läge der Mittelpunkt des Kreises im Ursprung
threshold	String-ParameterType	Nein	Definiert die Klasse, die als Schwellwertfunktion benutzt wird. Diese muss *GTAThresholdFunction* implementieren. Für alle Einheiten (lokal und zentral) muss dieselbe Schwellwertfunktion definiert sein

7 Web Anomaly Detection System (WADS)

7.1 Einleitung

Das Internet hat in der heutigen Gesellschaft einen enormen Stellenwert. Facebook, StudiVZ, flickR und Twitter sind nur einzelne Beispiele für Webprojekte, denen wir täglich begegnen und die wir in vielen Fällen auch alltäglich verwenden. Besonders deutlich wird dieser Einfluss, wenn man sich bewusst macht, dass diese Plattformen auch außerhalb des Internets thematisiert werden: Das Web 2.0 wirft Fragen nach Internetrecht und Datenschutz auf, die in der Politik stark diskutiert werden und den neu aufkommenden Webplattformen damit einen Platz auf den Titelseiten der Tageszeitungen sichern.
Solche Diskussionen beziehen sich häufig ausschließlich darauf, welcher Teil der gespeicherten personenbezogenen Daten öffentlich zugänglich gemacht werden darf. In den wenigsten Fällen beschäftigen sich die Medien mit den Sicherheitssystemen der Daten sammelnden Unternehmen. Das dieser Aspekt wichtig ist, zeigen z.B. die Statistiken des *Web Application Security Consortiums* von 2008 [20]: Dort wurden mit einer Wahrscheinlichkeit von 80% - 96% in 12186 aller untersuchten Webanwendungen Schwachstellen gefunden.
Die Arbeit von Kruegel und Vigna [30] beschäftigt sich mit der Frage, wie sich Webanwendungen schützen lassen. Anhand welcher Merkmale kann ein normaler Benutzer von einem Angreifer unterschieden werden? Wie kann ein Server feststellen, wann ein Angriff vorliegt und entsprechende Maßnahmen ergreifen? Was könnte an einer HTTP-Anfrage auffällig sein? Der Ansatz von Kruegel, Vigna und Robertson bewertet dabei jeden HTTP-Request anhand unterschiedlicher, getrennter Aspekte. Beispiele für solche Aspekte wären zum einen, dass sich bei Angriffen die Verteilung der Zeichen in einer HTTP-Anfrage deutlich von den üblichen Anfragen unterscheiden könnten, oder zum anderen, dass während automatisierter Angriffe Anfragen an einen Webserver in deutlich kürzeren Zeitabstände erfolgen, als bei gewöhnlichen Interaktionen von menschlichen Benutzern.
Diese verschiedenen Aspekte spiegeln sich bei Kruegel und Vigna in einzelnen Modellen wieder, die auf normalem Verhalten trainiert werden. Wenn eine HTTP-Anfrage von einem gelernten Modell abweicht, liefert das Modell einen *Score* zurück, dessen Wert die potentielle Bedrohung wiederspiegelt. Der Hintergrund dafür ist, dass sich gezeigt hat, dass ein Angriff im Internet in der Regel nicht alleine durch ein einziges Merkmal deutlich wird, sondern sich durch Anomalien in unterschiedlichen Bereichen auszeichnen und somit nur durch die Summe der Abweichungen erkannt werden kann.

7.2 Abbildung auf das Framework

Da der Ansatz von Kruegel und Vigna vom Aufbau her einem typischen Data Mining-Prozess entspricht, fällt eine Abbildung auf das Framework nicht schwer. Die verschiedenen Schritte der Anomalieerkennung wurden auf Knoten unseres Frameworks aufgeteilt:

AuditConsole Der Prozess beginnt mit dem Mitschnitt von HTTP-Requests, welcher von der Projektgruppe mit Hilfe des *AuditConsole*-Projekts [8] realisiert wurde. Das Tool lässt sich auf Servern installieren, speichert alle HTTP-Anfragen, bereitet diese für Analysen auf und bietet verschiedenste Funktionen, wie zum Beispiel das Filtern von Requests nach Zeitabschnitten. Dabei werden die HTTP-Anfragen in *AuditEvents* verpackt.

AuditEventParser Der *AuditEventParser* kann nun auf die AuditEvents der *AuditConsole* zugreifen und diese parsen. Nach dem Parsen können so gezielt die einzelnen Bestandteile des AuditEvents, wie zum Beispiel die Parameter der Anfrage, angesprochen werden. An dieser Stelle werden die für das jeweilige Modell relevanten Daten extrahiert.

AuditEventRouter Der *AuditEventRouter* verteilt die AuditEvents nun an die einzelnen Modellen und weißt ihnen eindeutige IDs zu.

Modelle An dieser Stelle fließen nun die Daten der einzelnen HTTP-Requests in die Modelle ein. Wie und welche Modelle hier verwendet werden, bleibt dem Anwender überlassen. Die zur Verfügung stehenden Modelle werden im nächsten Abschnitt aufgelistet und ausführlich erklärt. Beispielhaft könnte man sich vorstellen, dass hier alle Modelle parallel geschaltet werden, d.h. alle Modellknoten die gleichen AuditEvents empfangen, diese individuell auf Anomalien hin bewerten und einen *Score* ausgeben.

Score Die Score klasse dient als Interface für die Übergabe der Model-Ausgaben. Ein Score besitzt neben dem eigentlichen Wert, auch ein Gewicht und eine Zuversicht oder Confidence. Die Confidence kann beispielsweise bei probabilistischen Verfahren werdendet werden. Schlägt es fehl, oder bestimmte Teile, wird die Confidence herabgesetzt. Die Confidence sollte Werte zwischen 0 und 1 enthalten, wobei 1 starke Zuversicht ausdrückt.

[8] http://www.jwall.org

ModelAggregator Der *ModelAggregator* erhält nun als Eingabe die einzelnen Scores aller Modelle. Ausserdem wird über den *model-count* Parameter die Anzahl der Modelle, die zusammengefasst werden, festgelegt. Die Scores werden immer zusammen mit einer eindeutigen Audit-ID versendet. Hat der ModelAggregator nun entsprechend viele Scores zu einer Audit-ID gesammelt, sendet er diese an den ModelEvaluator.

ModelEvaluator Im ModelEvaluator, werden die gesammelten Scores ausgewertet. Beispielsweise kann der Durchschnitt der Scores gebildet werden, oder die gewichtete Summe. Jeder ModelEvaluator erbt von *AbstractModelEvaluator*. Dort ist bereichts alles wichtige implementiert. Insbesondere hat jeder AbstractModelEvaluator einen Parameter *threshold* der einen Schwellwert vorgibt. Das ist sehr nützlich das erwartet wird, das nach einem gewissen Verfahren ein *GesamtScore* gebildet wird, und sobald dieser den Schwellwert überschreitet, wird eine Anomaly erkannt. Dieser GesamtScore wird in der zu überschreibenden Methode *anomalyDetected* berechnet. Der Rückgabewert ist ein boolean der angibt ob eine Anomaly erkannt wurde oder nicht. Um einnen eigenen ModelEvaluator zu implementieren, muss lediglich von AbstractModelEvaluator geerbt werden und die anomalyDetected überschrieben werden. Im Framework sind *WeightedSumEnsembleConfThreshold* und *WeightedSumModelEvaluation*. implementiert. WeightedSumEnsembleConfThreshold bekommt einen Parameter *confThreshold* übergeben. Dieser enthält einen Wert von 0 bis 1. Dieser Wert bestimmt, den Schwellwert für die Confidence, der angibt ab wann ein Modell in die Berechung eingeht. Dieser ModelEvaluator bildet also die gewichtete Summe der Modelle, die eine gewisse Confidence bestitzen. *WeightedSumModelEvaluation* bildet die gewichtete Summe der Scores, wobei sowohl das eigentliche Gewicht, als auch dei Confidence als Faktoren dienen.

AlarmListener Falls ein Alarm ausgelöst wurde, hat der *AlarmListener* nun die Aufgabe zu entscheiden, was jetzt passiert. Denkbar wäre hier zum Beispiel dafür zu sorgen, dass weitere Anfragen, die von der IP des Angreifers kommen, geblockt werden.

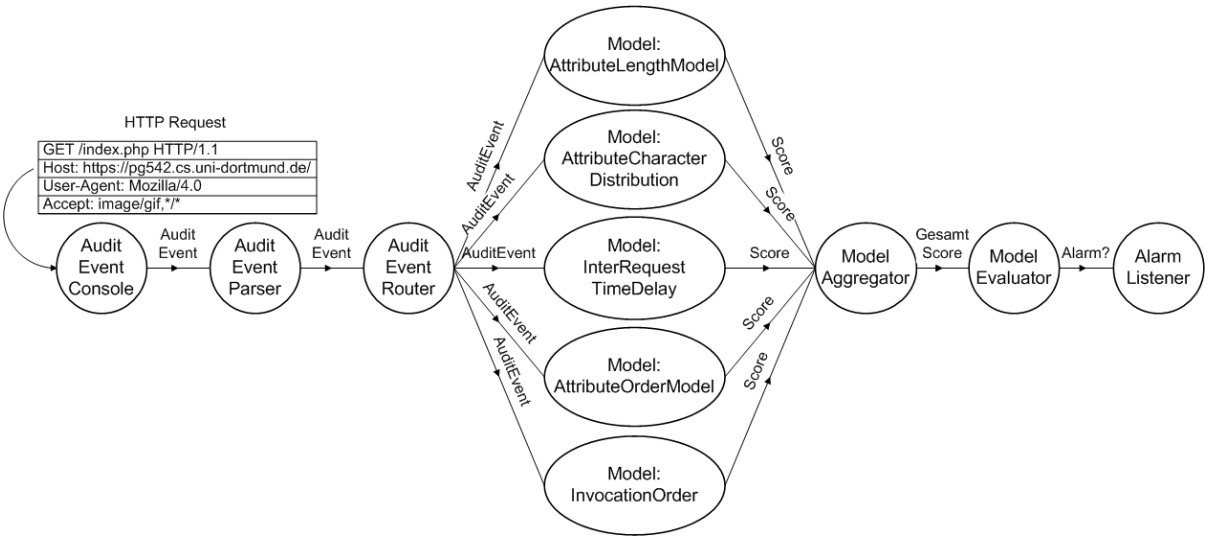


Abbildung 87: Möglicher Aufbau des Frameworks im WADS Anwendungsfall

In Abbildung 88 ist der in Abbildung 87 dargestellte Aufbau des WADS Anwendungsfalls als XML-Konfiguration zu sehen.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <nodenetwork xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:noNamespaceSchemaLocation="../../main/resources/
       schema.xsd">
3    <version>0.2.0</version>
4    <namingservicefactory class="edu.udo.cs.pg542.core.naming.NamingServiceFactory"/>
5    <assign id="wads" adress="127.0.0.1" />
6
7    <container id="wads">
8        <node class="edu.udo.cs.pg542.wads.node.input.AuditEventParser" id="audit-event-parser" kickoff="true">
9          <parameter name="file" value="/home/share/pg542-audit.log" type="String" />
10         <parameter name="startAtBeginning" value="true" type="Boolean" />
11         <link>
12           <target-id>audit-event-router</target-id>
13         </link>
14     </node>
15     <node class="edu.udo.cs.pg542.wads.node.preprocessing.AuditEventRouter" id="audit-event-router">
16       <link>
17         <target-id>model-attribute-length</target-id>
18         <target-id>model-attribute-character-distribution</target-id>
19         <target-id>model-inter-request-time-delay</target-id>
20         <target-id>model-attribute-order</target-id>
21         <target-id>model-invocation-order</target-id>
22       </link>
23     </node>
24     <node class="edu.udo.cs.pg542.wads.node.model.AttributeLengthModel" id="model-attribute-length">
25       <parameter name="weight" type="Double" value="1.0"/>
26       <link><target-id>model-aggregator</target-id></link>
27     </node>
28     <node class="edu.udo.cs.pg542.wads.node.model.AttributeCharacterDistributionNode" id="model-attribute-character-
           distribution">
29       <parameter name="weight" type="Double" value="1.0"/>
30     <link><target-id>model-aggregator</target-id></link>
31     </node>
32   <node class="edu.udo.cs.pg542.wads.node.model.InterRequestTimeDelay" id="model-inter-request-time-delay">
33     <parameter name="weight" type="Double" value="1.0"/>
34     <link><target-id>model-aggregator</target-id></link>
35     </node>
36     <node class="edu.udo.cs.pg542.wads.node.model.AttributeOrderModel" id="model-attribute-order">
37       <parameter name="weight" type="Double" value="1.0"/>
38     <link><target-id>model-aggregator</target-id></link>
39     </node>
40     <node class="edu.udo.cs.pg542.wads.node.model.InvocationOrder" id="model-invocation-order">
41       <parameter name="weight" type="Double" value="1.0"/>
42     <link><target-id>model-aggregator</target-id></link>
43     </node>
44     <node class="edu.udo.cs.pg542.wads.node.postprocessing.ModelAggregator" id="model-aggregator">
45       <parameter name="model-count" type="Integer" value="5"/>
46       <link><target-id>model-evaluator</target-id></link>
47     </node>
48     <node class="edu.udo.cs.pg542.wads.node.postprocessing.ModelEvaluator" id="model-evaluator">
49       <parameter name="threshold" type="Double" value="0.9"/>
50       <link><target-id>alarm-listener</target-id></link>
51     </node>
52     <node class="edu.udo.cs.pg542.wads.node.output.AlarmListener" id="alarm-listener" />
53   </container>
54 </nodenetwork>
```

Abbildung 88: XML-Konfiguration des WADS Anwendungsfalles

7.3 Modelle

7.3.1 Attribute Length

Bei diesem Modell werden die einzelnen Attribute eines HTTP-Requests betrachtet. Hierbei sind lediglich die Längen der Attributwerte und nicht die Werte selbst von Interesse. In Anlehnung an [30] wird angenommen, dass außergewöhnlich lange Werte potentielle Angriffe bedeuten. Dieses Modell geht davon aus, dass keine Angriffe durch das Verkürzen von Attributwerten - also dem Weglassen von Zeichen in einer Zeichenkette - möglich sind. Im unten stehenden Beispiel wird diese intuitive Annahme verdeutlicht: Während die normale Suchanfrage in Zeile 1 keinerlei Schaden anrichten würde, wenn der Wert '42' verkürzt oder weggelassen wird, ist es offen-

sichtlich, dass ein längerer Wert wie in Zeile 2 auf einen Angriff hinweisen kann.

```
1 http://webserver/cgi-bin/find.cgi?ID=42
2 http://webserver/cgi-bin/find.cgi?ID=42+UNION+SELECT+login,+password,+'x'+FROM+user
```

Aus diesem Grund ist es wünschenswert für jeden neuen Wert eines Attributs entscheiden zu können, ob dieser Wert tolerierbar ist, oder aber eine Länge besitzt, die auf einen möglichen Angriff schließen lässt. Während das Modell von Kruegel, Vigna und Robertson [30] in einer vorgelagerten Trainingsphase die zugrundeliegende Verteilung durch das Bestimmen von Mittelwert und Varianz anzunähern versucht, bestimmt dieses Modell eine maximale Länge für jedes Attribut. Hierzu wird dem Modell durch einen Parameter ϕ der angenommene Anteil gutartiger Anfragen übergeben. Dies bedeutet, dass $(1 - \phi) \cdot 100\%$ aller Anfragen als schadhaft geschätzt werden. Zur Verwaltung solcher Rangstatistiken bieten sich Quantilalgorithmen an. Da ein inkrementelles Verfahren für schnelle Datenströme benötigt wird, dient der Algorithmus **Ensemble Quantiles** (Kapitel 6.2.4) als Basis für das Modell zur Überprüfung von Attributlängen.

In einem Vorverarbeitungsschritt wird jeder HTTP-Request in seine einzelnen Attribute aufgeteilt. Da die Anzahl von möglichen Attributen überschaubar ist, wird für jedes Attribut eine Instanz von Ensemble Quantiles verwaltet. Die Werte der einzelnen Attribute werden durch ihre Längen ersetzt und dienen als Eingabe für die jeweilige Instanz des Quantilalgorithmus. Sobald für ein Attribut ausreichend viele Beispiele verarbeitet wurden, können neue Werte dieses Attributs beurteilt werden. Hierzu wird ϕ verwendet um eine Anfrage an die Instanz des Ensemble Quantiles Algorithmus zu stellen, welche die Längen für dieses Attribut verwaltet.
Das Ergebnis dieser Anfrage ist die obere Schranke für die Länge sicherer Attributwerte. Während beim Ansatz in [30] für jeden Wert eines Attributs der Abstand zum Mittelwert dieses Attributs und die Wahrscheinlichkeit für diese Abweichung mit der Chebyshev-Ungleichung bestimmt werden muss, genügt es hier die neue Länge mit der maximalen Länge zu vergleichen. Dadurch ist es zwar nicht mehr möglich das Gefährdungspotential eines Attributwerts differenziert zu bestimmen - die Entscheidung ist nun binär - jedoch gewinnt die Testphase dadurch an Performance und eignet sich somit zur Verarbeitung von schnellen Datenströmen.

Neben dem Gewinn an Performance ist der inkrementelle Charakter dieses Modells ein Hauptvorteil gegenüber dem Model von Kruegel, Vigna und Robertson [30]. Dadurch wird das Modell zur Überprüfung von Attributlängen von einem offline zu einem online Lernalgorithmus. Hiermit wird gewährleistet, dass ein Drift in den Daten erkannt und das Modell angepasst wird.
Gegenüber einer statischen Trainingsphase auf sauberen Daten besteht bei einer kontinuierlichen Trainingsphase allerdings die Gefahr, dass auch Angriffe in das Modell einfließen. Allerdings sollten Angriffe Ausnahme und nicht Regel sein und so-

mit ein viel kleineres Volumen des Datenstroms ausmachen. Außerdem dient dieses Modell zur Erkennung von Angriffen, die signifikant längere Attributwerte verwenden (*Code/SQL-Injection*, *Buffer-overflow*). Es folgt also, dass bei Angriffen verwendete Werte in Quantilen des Bereichs $(\phi, 1]$ zu finden und somit vernachlässigbar sind.

Die Ausgabe des Modells zur Überprüfung der Attributlänge ist wie bei allen WADS-Modellen ein Score. Weil eine HTTP-Anfrage nicht als Ganzes betrachtet, sondern in seine einzelnen Attribute zerlegt wird, müssen die einzelnen Scores für die Attribute zu einem Score für die gesamte HTTP-Anfrage aggregiert werden. Das Modell geht hier davon aus, dass bereits ein schadhaftes Attribut genügt um die gesamte HTTP-Anfrage als potentiellen Angriff zu kennzeichnen. Es wird also kein durchschnittlicher Wert für den Score gebildet, sondern das Maximum aller einzelnen Scores verwendet. Dieser Wert ist wegen dem binären Charakter der Testphase entweder 0 oder 1.
Die Konfidenz eines Scores ist vorgegeben durch die Parametrisierung von Ensemble Quantiles. Wie jeder approximative Quantilalgorithmus wird auch hier ein maximaler Fehler ϵ vorgegeben. Dieser Wert ϵ dient als Konfidenz für die erzeugten Scores.

7.3.2 Attribute Character Distribution

Das Modell Attribute Character Distribution versucht anhand der Zeichenverteilung der Parameter eines HTTP-Requests einen Angriff zu erkennen. Die Idee ist dabei, dass sich diese Verteilungen bei Angriffen stark von den Zeichenverteilungen bei normalen HTTP-Anfragen unterscheiden. Eine normale Anfrage könnte zum Beispiel, wie in Zeile 1 unten aussehen: Ein Skript in der Datei *navigate.php* bekommt als Parameter ein Verzeichnis (hier: */tmp/index.php*) und ein Schlüsselwort (*PG*) übermittelt. In Zeile 2 dagegen sieht man, wie die URL des HTTP-Requests eines Angreifers aussehen könnt: Dieser versucht mittels *Directory Traversal* auf Systemdateien des Servers zuzugreifen, in dem er typische Systempfade als Verzeichnisse ausprobiert. Im zweiten Parameter sieht man einen typischen Parameter für eine *SQL-Injection*. Hier wird neben der Übertragung des Schlüsselwortes *PG* versucht ein zweites SQL-Statement auszuführen, welches dem Account des Angreifers Administrationsrechte in der Webanwendung zuweist.

```
1 https://pg542.cs.uni-dortmund.de/trac/navigate.php?dir=/tmp/index.php&keyword=PG
2 https://pg542.cs.uni-dortmund.de/trac/navigate.php?dir=/../../../../../root/system&keyword=PG;UPDATE+USER+SET+TYPE="admin"+
      WHERE+ID=23
```

Training

Im Training werden nun diese Schlüssel-Wert-Paare der Parameter des HTTP-Requests betrachtet. Dabei werden für jeden Schlüssel einzeln die Wertedaten in ASCII-Zahlen übersetzt. Dies könnte zum Beispiel für */tmp/index.php* wie folgt aussehen: 47 116 109

112 47 105 110 100 101 120 46 112 104 112. Daraufhin werden die absoluten Häufigkeiten der ASCII-Codes gezählt, in relative Häufigkeiten umgewandelt, aufsteigend sortiert und in ein Histogramm (*Character Distribution (CD)*) eingetragen.

Abbildung 89: *Character Distribution (CD)* eines Parameterwertes

Das häufigste ASCII-Zeichen steht jetzt zusammen mit seiner relativen Häufigkeit hinten, wodurch allerdings der Bezug zu dem konkreten ASCII-Zeichen verloren geht. Dieser Informationsverlust ist für das weitere Verfahren jedoch irrelevant, da nur die Verteilungen der Zeichen miteinander verglichen werden. Für jeden Schlüssel wird die sogenannte *Idealized Character Distribution (IDC)* als Modell verwaltet. Wenn bereits ein Modell für eine neue Character Distribution besteht, wird dieses Modell aktualisiert. Für die Aktualisierung wird die CD geeignet gewichtet und mit der ICD verrechnet. Falls noch keine ICD für den Schlüssel vorliegt, wird die CD zur neuen ICD des Schlüssels.

Test

Im Test werden die ersten Schritte analog zum Training durchgeführt: Das Schlüssel-Wert-Paar wird aus dem HTTP-Request entnommen und eine Character Distribution berechnet. Falls nun eine Idealized Character Distribution existiert, werden diese beiden Verteilungen miteinander verglichen. Dies wird mit dem χ^2-Test durchgeführt, wozu die Verteilungen in 6 sogenannten Bins komprimiert werden. In dieser Implementierung werden die möglichen 256 ASCII-Zeichen in folgende Bins eingeteilt: [255,16],[15,12],[11,7],[6,4],[3,1],[0]. Das heißt, das häufigste Zeichen (das an der letzten Stelle im Histogramm steht) füllt ein eigenes Bin, bei allen anderen Zeichen werden die Bins gebildet, in dem die Mittelwerte über die Bin-Intervalle berechnet werden.

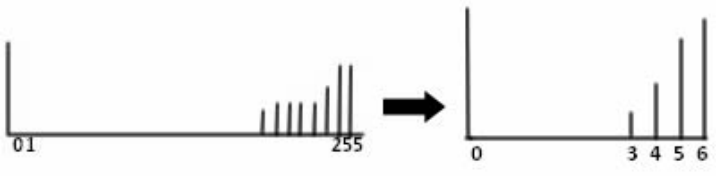


Abbildung 90: Transformation in Bins

Über die Bins von CD und ICD wird danach der χ^2-Wert berechnet und mit Hilfe einer implementierten Tabelle, die der χ^2-Verteilung mit fünf Freiheitsgeraden entspricht, das $1 - \phi$ Quantil an der Stelle χ^2 berechnet.

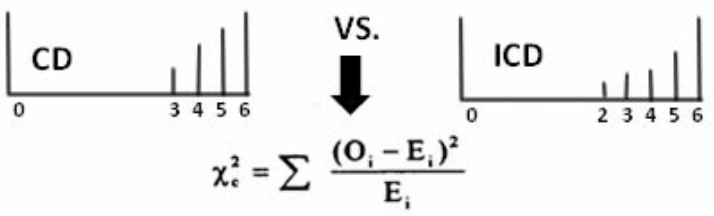


Abbildung 91: Berechnung des χ^2-Wertes aus den Bins von CD und ICD

Der relative Wert am Ende der Rechnung gibt die Unähnlichkeit zwischen den beiden Verteilungen an und wird als Score ausgegeben.

7.3.3 Token Finder

Für bestimmte Parameter in einer Webseiten-Anfrage sind oft nur Werte aus einer endlichen Domäne erlaubt. Dies ist insbesondere der Fall, wenn bei einem Formular Werte aus einer Dropdown-Liste verschickt werden. Bei solchen Attributen kann man einen Angriff dann erkennen, wenn abweichende Werte vorkommen.

Die Aufgabe bei diesem Modell besteht darin, zu lernen, ob ein Parameter aus einer begrenzten Domäne kommt oder beliebige Werte annehmen kann. Wenn es sich um beliebige Werte handelt, dann kann das Modell keine Aussage treffen, so dass als Score-Wert eine 0 zurückgegeben wird. Bei einer festen Domäne wird überprüft, ob der vorliegende Wert in dieser Domäne liegt. Handelt es sich um einen Wert, der nicht bekannt ist, dann gibt das Modell 1 als Score-Wert aus.

Dieses Modell benötigt also zunächst eine Trainingsphase in der auf Nicht-Angriffen gelernt wird, ob es sich um eine Domäne oder um beliebige Werte handelt. Wenn die Anzahl der verschiedenen Werte proportional mit der Anzahl der einkommenden Werte steigt, dann handelt es sich wahrscheinlich, um zufällige Werte. Diese Zusammenhang kann man mit zwei Funktionen f und g darstellen, wobei f linear mit jedem neuen Element im Datenstrom steigt und g entweder steigt, wenn wir ein neuen Wert im Strom finden, oder sinkt.

Wenn wir nun die Varianz und Kovarianz für diese Funktionen f und g berechnen, dann können wir auch die Korrelation ρ dieser Funktionen bestimmen.

$$\rho = \frac{Covar(f,g)}{\sqrt{Var(f) * Var(g)}}$$

Sind diese beiden Funktion negativ korreliert ($\rho < 0$), dann kann das Modell von einer begrenzten Domäne ausgehen.

Die Lernphase beschränkt sich also auf die Berechnung dieser Varianz und Kovarianz, die sich mit Hilfe des Verschiebungssatzes auch auf einem Stream ermitteln lässt.

In dieser Zeit müssen natürlich die einkommenden Daten gespeichert werden, um zu ermitteln, ob der Wert schon einmal aufgetaucht ist. Dies kann alternativ mit einem approximativen Zählalgorithmus realisiert werden. Spätestens nach der Lernphase werden aber nur noch die Werte aus den festen Domänen benötigt. Da Domänen i.d.R. klein sind, ist der entstehende Speicheraufwand vertretbar.

7.3.4 Attribute Presence Or Absence

Viele Anfragen an einen Server werden nicht direkt durch den Nutzer, sondern oftmals durch unterstützende Software erstellt. Der Nutzer übergibt dabei lediglich Parameter wie z.B. Nutzernamen und Passwort an ein Programm welches diese Eingaben weiterverarbeitet und eine URI daraus erstellt. Beispiele für solche Programme sind Browser die einen Link verfolgen, oder Mail-Programme mit vordefinierter Anfrage-Syntax.

Da die URIs von Programmen erzeugt werden, welche einer vordefinierten Syntax folgen, ähneln sich die URIs untereinander sehr. Beispielsweise wird ein Browser die Anzahl und die Reihenfolge der übergebenen Parameter innerhalb der URI niemals verändern. Aus den angefragten URIs lässt sich somit ein Muster ableiten, nach welchem die Anfragen aufgebaut sind.

Ein Angreifer erstellt seine URIs meist händisch und wird dabei von dem statischen Muster eines anfragenden Programms abweichen. Idee dieses Modells ist es daher, aus einer Reihe von angefragten URIs Muster abzuleiten und diese mit jeder neuen Anfrage abzugleichen. Ist das angefragte Muster bereits bekannt, kann ein Angriff aus Sicht des Attribute Presence Modells ausgeschlossen werden. Ist der Aufbau der aktuellen URI aber gänzlich unbekannt oder nur sehr selten aufgetreten, könnte es sich um einen potenziellen Angriff handeln.
Zu beachten ist hierbei, dass sich die Anfrage-Syntax eines Programms mit der Zeit natürlich verändern kann. Außerdem wäre es möglich, dass neue Programme zur Generierung der Anfragen verwendet werden, die ein abweichendes Anfrage-Muster verwenden. Daher sollte die Lernphase dieses Modells auf Basis eines *Sliding Window* durchgeführt werden. Damit ist sichergestellt, dass Veränderungen in den validen Anfragen zu einer Veränderung des Modells führen und somit die Anzahl der Fehlalarme reduziert wird.

Das Modell wurde innerhalb des Anwendungsfalls auf zwei unterschiedliche Arten umgesetzt. Eine Implementierung erstellt innerhalb der Lernphase eine Baumstruktur aus den vorkommenden URI-Parametern. Für jeden Parameter wird hierbei während der Lernphase an der entsprechenden Stelle im Baum ein Knoten erzeugt. In den Blättern werden jeweils die Häufigkeiten dieses Pfades und damit des repräsentierten URI Musters gezählt. Innerhalb der Testphase wird nun versucht, den Baum anhand der in der URI übergebenen Parameter zu durchlaufen. Landet das Verfahren dabei

in einem Blatt, wird anhand der relativen Häufigkeit dieses URI Musters die Wahrscheinlichkeit für einen Angriff bestimmt. Erreicht das Verfahren beim Durchlaufen des Baums kein Blatt, ist die relative Häufigkeit 0, die Wahrscheinlichkeit für einen Angriff somit 1.

Die alternative Implementierung nutzt den in Kapitel 6.1.1 vorgestellten Ansatz *Lossy Counting*. Dabei werden die Parameternamen in der Reihenfolge ihres Auftauchens konkateniert und als Eingabewert für den Lossy Counting Algorithmus übergeben. Dieser gibt innerhalb der Testphase nun wiederum die relative Häufigkeit dieses URI Musters zurück, aus welcher die Wahrscheinlichkeit für einen Angriff abgeleitet wird.

Die berechneten Angriffs-Wahrscheinlichkeiten werden als Score zurückgeliefert und dienen im weiteren Verlauf des WADS zur Berechnung eines Gesamtscores, welcher eine Interaktion als Angriff oder Normalzustand klassifiziert.

7.3.5 Access Frequency

Webanwendungen bestehen häufig aus vielen kleineren Funktionen, die dem Benutzer bereit gestellt werden. Die Anzahl der Zugriffe eines Benutzers auf ein und dieselbe Funktion innerhalb eines Zeitintervalls sind maßgeblich von der Funktionalität selbst abhängig. Beispielsweise wird die Anzahl der Zugriffe auf die Login-Funktionalität einer Website pro Nutzer im Durchschnitt kleiner als zwei sein, wenn man von einem Zeitintervall von fünf Minuten ausgeht. Die Gesamtzahl der Zugriffe kann - abhängig von Frequenz von Nutzerinteraktionen mit der Website - innerhalb des gleichen Zeitintervalls jedoch um ein vielfaches höher sein. Dem gegenüber wird die Suchfunktion einer Website nur von einer geringen Anzahl von Nutzern innerhalb eines Zeitintervalls verwendet. Ein einzelner Nutzer wird jedoch häufig eine große Anzahl an Anfragen durch die Suchfunktion erzeugen. Somit ist der Verhältnis der Nutzerzugriffe pro Zeitintervall zur Gesamtanzahl der Zugriffe pro Zeiteinheit sehr hoch.

Eine plötzliche Veränderung der Zugriffswerte auf eine bestimmte Funktionalität, könnte einen möglichen Angriff als Ursache haben. Steigt beispielsweise das Verhältnis der Nutzerzugriffe zur Gesamtanzahl der Zugriffe rapide an, könnte dies auf einen Bruce-Force-Angriff hindeuten.

Die grundsätzliche Idee dieses Modells ist es daher, für jede serverseitig bereitgestellte Anwendung die durchschnittliche *Access Frequency*, die Zugriffsfrequenz, als Verhältnis zwischen Nutzerzugriffen pro Zeitintervall und Gesamtanzahl der Zugriffe pro Zeitintervall zu berechnen. Dazu ist es notwendig, innerhalb eines Zeitintervalls für jeden Nutzer alle Zugriffe auf eine bestimmte Anwendung zu zählen.

Innerhalb der Testphase des Modells, wird - genau wie in der Lernphase - nun die Zugriffsfrequenz eines jeden Nutzers pro Anwendung berechnet. Aus dem Verhältnis zwischen der erwarteten, durchschnittlichen Zugriffsfrequenz und der tatsächlich vorliegenden Zugriffsfrequenz kann nun die Wahrscheinlichkeit für einen Angriff ermittelt werden. Diese wird als Score zurückgegeben und dient zur Berechnung des Gesamtscores aller benutzten Modelle.

7.3.6 Inter-Request Time Delay

Das Modell betrachtet, gruppiert nach den IP-Adressen, die Zeiten zwischen den einzelnen Requests. Anhand der betrachteten Request-Zeit kann so erkannt werden, ob ein Angriff vorliegt oder nicht.
Mit Hilfe eines Baumes werden die einzelnen IP-Adressen dargestellt. Dies hat den Vorteil, dass IP-Adressen von Anfang an gruppiert werden.

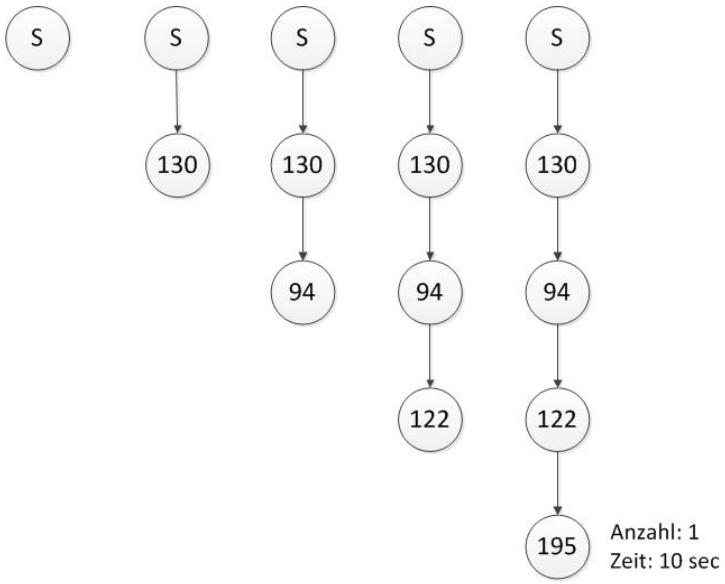


Abbildung 92: *Modell: Baum mit der IP-Adresse 130.94.122.195*

Bei der Erstellung eines Baumes bleibt die Wurzel als Root-Element erhalten, um so zwischen den einzelnen IP-Adressen direkt unterscheiden zu können. Nach dem Einfügen der ersten Adresse, werden zwei Arten des weiteren Einfügens betrachtet. Dabei wird zwischen einer IP-Adresse, die bereits - vollständig oder auch nur teilweise - vorhanden ist und einer vollkommen neuen IP-Adresse unterschieden. In beiden Fällen wird zunächst in der ersten Ebene des Baumes gesucht, ob der erste Wert der Adresse schon im Baum vorhanden ist. Ist dies nicht der Fall, so wird ein neuer Pfad mit der neuen IP-Adresse angelegt. Ist der erste Teil vorhanden, so werden die weiteren Ebenen betrachtet. Dies wird so lange fortgeführt bis die IP-Adresse komplett gefunden wurde. Anschließend muss die Anzahl um eins erhöht werden und es wird

anhand der χ^2- Verteilung ermittelt, ob es sich um eine ausschlaggebende Abweichung handelt und somit um ein potentieller Angriff vorliegt oder die Request-Zeit mit den bisherigen Antwortzeiten konform ist. Die Abweichung wird durch einen Score dargestellt. Je größer der Wert des Scores ist, umso wahrscheinlicher handelt es sich hierbei um einen Angriff. Wird die IP-Adresse nur in Teilen gefunden so wird der neue Teil, wie beim Einfügen einer neuen Adresse, angehängt. Um nicht eine Fülle an Adressen verwalten zu müssen, wird der Zähler mit jeder neu ankommenden Adresse mit einem Verfallswert multipliziert. So kann gewährleistet werden, dass Adressen die über einen längeren Zeitraum nicht betrachtet werden aus dem Modell herausfallen.

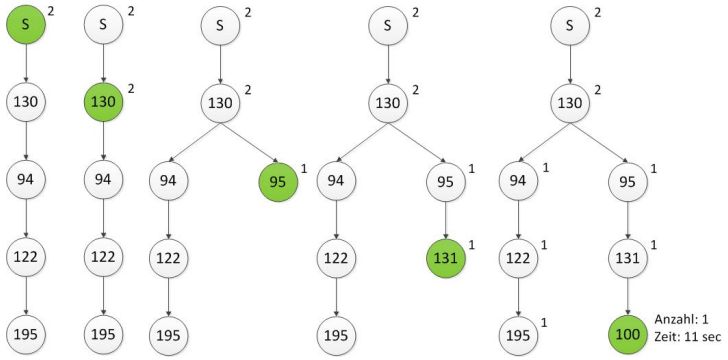


Abbildung 93: *Einfügen der IP-Adresse 130.95.131.100 zum bestehenden Modell*

7.3.7 Invocation Order

Eine moderne Webanwendung besteht in der Regel aus einer Menge unterschiedlicher Teilfunktionalitäten, deren Verwendung nur in einer bestimmten Reihenfolge sinnvoll ist. Schaut man sich zum Beispiel die Online-Shopping Plattform ebay [9] an, so kann man zum Beispiel feststellen, dass

1. ein Kauf nur nach Anmeldung mit einem gültigen Nutzeraccount möglich ist,
2. der Kauf eines Artikeln nur nach Betrachtung des Artikels ermöglicht wird und
3. Kontodaten nur nach dem Kauf eingesehen werden können.

Wird diese vorgegebene Reihenfolge gebrochen, handelt es sich nach [30] sehr wahrscheinlich um eine Anomalie. Dies könnte beispielsweise der Fall sein, wenn ein Angreifer versucht die Login-Seite zu umgehen, um direkt auf vertrauliche Systemkomponenten zuzugreifen. Diese soll von der Invocation Order Modell Komponente des *Web Anomaly Detection Systems* erkannt werden. Da eine gültige Aufrufreihenfolge nicht in Form von Regeln vom Benutzer eingegeben werden soll, muss das System

[9] http://www.ebay.com

diese lernen. Dies erfolgt nicht wie in [30] beschrieben in einer dedizierten Lernphase, sondern parallel zur Anomaliedetektion. Hierzu wird auf Basis der Session ID und der angefragten Systemressource (URL) aus den AuditEvents ein nichtdeterministischer Automat generiert, der Übergänge zwischen den angefragten Systemressourcen inklusive der Häufigkeit ihres Auftretens repräsentiert. Häufig auftretende Übergänge werden im folgenden als erlaubte Zugriffe bewertet, wohingegen sehr sporadisch oder gar nicht repräsentierte Übergänge als tendenzielle Angriffe (Anomalie) erkannt werden. Basierend auf dieser Bewertung liefert das Modell für ein AuditEvent einen *Anomaly-Score* bestehend aus einer reellen Zahl aus dem Wertebereich zwischen 0 und 1. Ein Wert von 0 bedeutet hierbei, dass das Event im Bezug auf die Invocation Order ungefährlich ist.

7.3.8 Evaluation

Aufbau des Experiments

Für die Evaluation des *Web Anomaly Detection Systems* wurde ein Experiment mit der *Test-and-Train*-Methode erstellt. Die Projektgruppe hat sich gegen eine vorgeschaltete Trainingsphase entschieden, da zu erwarten ist, dass sich die Modelle bei Webanwendungen mit der Zeit verändern. Außerdem kann in der Praxis im allgemeinen keine Trainingsphase mit ausschließlich gutartigen Anfragen angenommen werden. Da das Modell *Token Finder* jedoch eine Trainingsphase voraussetzt, wurde für dieses Modell eine Trainingsphase mit 25000 Elementen verwendet.
Der Datensatz enthält einen Teil der Anfragen, die die Projektgruppe an die intern verwendeten Server gestellt haben. Im Detail wurden log files der bug tracking Anwendungen *trac* und *MantisBT* erstellt. Diese log files wurden zu einem Datensatz mit insgesamt 53755 unterschiedlichen Ereignissen zusammengefasst.
Zu Beginn des Datensatzes sind 29709 authentische Anfragen, die von den Teilnehmern der Projektgruppe während der täglichen Arbeit mit dem System entstanden sind. In dieser Phase liegen also keine Angriffe vor. Anschließend folgen 24046 Einträge, die ausschließlich Angriffe enthalten. Zu diesem Zweck wurde das *w3af (Web Application Attack and Audit Framework)* [10] verwendet um typische Angriffsszenarien zu simulieren.
Durch die Verwendung des w3af Frameworks war es besonders einfach Angriffe von legitimen Einträgen im log file zu unterscheiden, da sich der *User-Agent* in Einträgen von Angriffen als w3af.sourgeforge.net ausgibt und sich somit klar von gewöhnlichen User-Agents abgrenzt. Hierdurch wird es möglich, sowohl den relativen Anteil richtig erkannter Angriffe (*true positives*) als auch die Rate der fälschlich als Anomalie erkannten Angriffe *false positives* für jedes Modell zu bestimmen.
Jedes Modell extrahiert die benötigten Informationen aus den Rohdaten des log files. Hierzu zählen je nach Modell die *SessionID, Request-Parameter* und *IP-Adressen*. Der

[10] http://w3af.sourceforge.net/

als Label verwendete Parameter *User-Agent* wird von keinem Modell zum Lernen oder zur Anomalie-Erkennung verwendet.

Attribute Length Model

Das Attribute Length Model hat während unserer Tests, wie bereits in Kapitel 7.3.1 vermutet, Angriffe erkannt, die sich durch das Einfügen von Befehlen für unterliegende Anwendungen (wie das Betriebssystem, SQL-Anwendungen etc.) manifestieren (Abbildung 94). Hierzu zählen *HTML-, Code-* und *SQL-Injections* sowie *Directory-Traversal* Angriffe.
Als Beispiel für einen als Angriff erkannten Parameter sollen hier die unten aufgelisteten Parameter eines *AuditEvents* dienen. Von den drei Parametern des AuditEvents (*password, perm_login* und *username*) ist der Parameter *password* auffällig. Das Attribut Lenght Model hat in der ersten Phase Werte für den Parameter *password* verarbeitet, die für gewöhnlich Längen zwischen sechs und acht Zeichen hatten. Der nun auftretende Parameter „%3B%2Fusr%2Fsbin%2Fping+-s+localhost+1000+10+“ ist somit deutlich länger und aus Sicht des Modells verdächtig. Diese Einschätzung ist für diesen Fall auch zutreffend, da sich hinter diesem Wert der Versuch verbirgt, das Programm „*ping -s localhost 1000 10*“ im Verzeichnis „/usr/sbin/“ auszuführen.

```
1 <password; \%3B\%2Fusr\%2Fsbin\%2Fping+-s+localhost+1000+10+>
2 <perm\_login; >
3 <username; WBkQGVz>
```

Bei den fälschlicherweise als Anomalie erkannten Anfragen (false positives), handelt es sich hauptsächlich um Freitextfelder. Das Modell ist für diese Art von Parameteren nicht geeignet, weil hier eine besonders große Streuung auftritt.
Des Weiteren wurde festegestellt, dass das Attribute Length Model bei Parametern mit festen Auswahlmöglichkeiten fälschlicherweise auf Anomalien schließen kann. Fehlalarme treten an dieser Stelle auf, wenn längere Werte selten verwendet werden.

Bei der Betrachtung der Erkennungsrate des Attribute Length Model fällt zunächst auf, dass insgesamt nur knapp 250 Anomalien entdeckt wurden. Die Rate der false positives liegt mit 0.002 bei unter einem Prozent. Demgegenüber fällt allerdings auf, dass die Rate der korrekt erkannten Anomalien unter allen Angriffen auch nicht sehr hoch ist. Dieser Anteil hält sich relativ konstant zwischen 12 und 14%. Die niedrige false positive Rate lässt zwar prinzipiell einen Einsatz als Intrusion Detection bzw. Intrusion Response Systems zu, ist allerdings aufgrund der geringen Erkennungsrate nur als Teil eines größeren Systems zu gebrauchen.

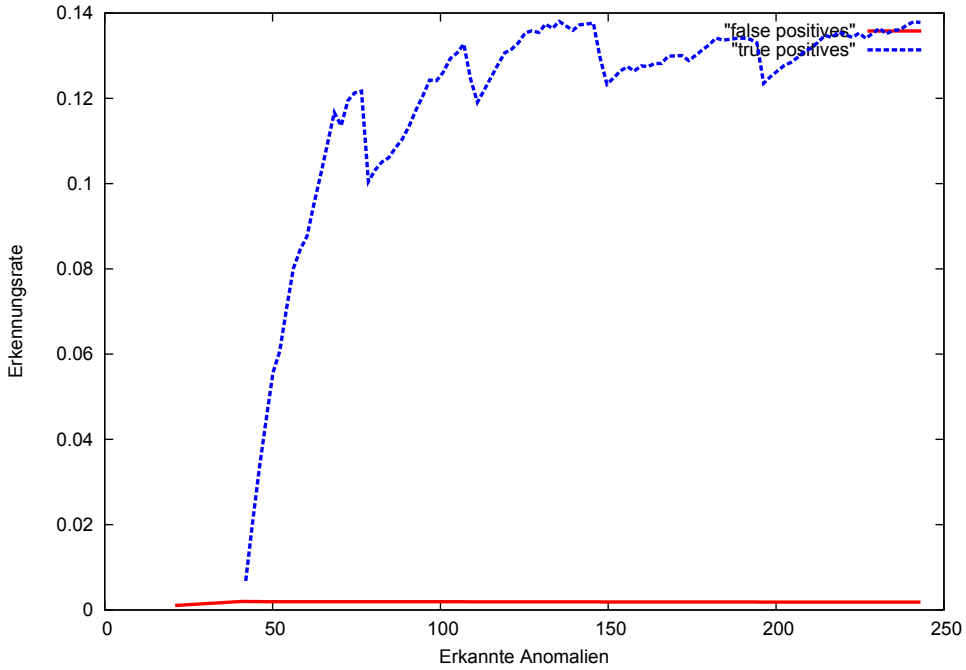


Abbildung 94: Rate der richtig und falsch erkannten Anomalien

Attribute Character Distribution

Mit dem *Attribute Character Distribution Model* wurden deutlich mehr Anomalien als vom *Attribute Length Model* erkannt. Die erkannten Anomalien sind auch hier wieder durch Angriffe ausgelöst worden, die URL Escape-Sequenzen verwenden. Hierzu zählen *SQL-, Code-* und *HTML-Injections* sowie *Directory-Traversal* Angriffe. Eine Auffälligkeit des Attribute Character Distributions Models ist, dass viele Angriffe als Anomalie gekennzeichnet wurden, weil das verwendete Angriffstool für Parameter wie z.B. Benutzernamen und Passwörter *leetspeak*[40] verwendt. Dadurch erkennt dieses Modell vorbereitete Schritte für einen Angriff durch das w3af Frameworks. In unten stehendem Beispiel sollten per *brute force* valide Benutzernamen gefunden werden. Der Angriff fiel sowohl durch das Verwenden von *„w3af-FrAmEW0rK.“*, als auch der zufälligen Zeichenkette *„AUwlWZr“* auf.

```
1 <password; w3af-FrAmEW0rK.>
2 <return; my_view_page.php>
3 <username; AUwlWZr>
```

Bei der Analyse der irrtümlich als Anomalie gekennzeichneter AuditEvents fiel auf, dass besonders Passwörter und Benutzernamen Fehlalarme auslösten. Bei diesen Parametern ist die Streuung besonders groß, da einige Werte aus 2-3 Zeichen bestehen die mehrfach verwendet werden und andere Werte völlig gleichverteilt sind und somit 6-10 unterschiedliche Zeichen besitzen.
Darüber hinaus haben Freitextfelder oft zu false positives geführt. Dies ist allerdings darauf zurückzuführen, dass diese Felder in den wenigsten Fällen sinnvoll in der

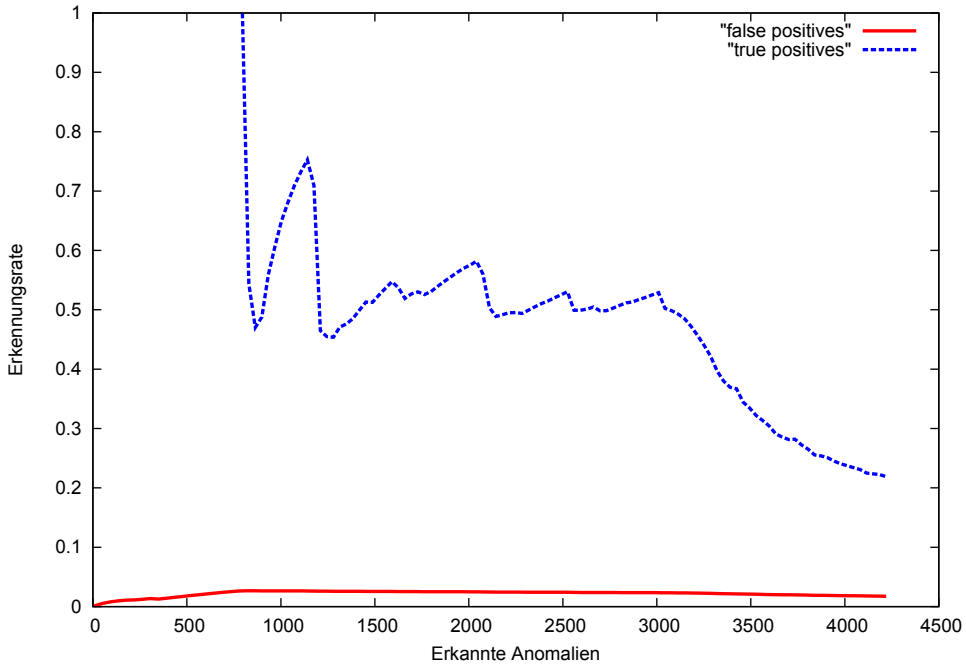


Abbildung 95: Rate der richtig und falsch erkannten Anomalien

Trainingsphase ausgefüllt wurden. Wenn im Anschluss längere Texte für diesen Parameter verwendet wurden, weicht dies stark von der Idealized Character Distribution ab.

Wie bereits erwähnt, erkennt dieses Modell deutlich mehr Anomalien als das Attribute Length Model. Dadurch sind auch im wesentlichen die höheren Raten der true positives und false positives in Abbildung 95 zu erklären. Die Rate der false positives ist mit 2 bis 3% auch bei diesem Modell moderat, es werden jedoch wichtige Benutzerinteraktionen wie das Einloggen als Anomalie erkannt. Der Verlauf der true positive Rate zeigt zu Beginn der Angriffe eine hohe Erkennungsrate von 50 bis 70%. In diesem Teil des log files fanden viele angriffsvorbereitende Schritte statt in denen brute force Angriffe erkannt wurden. Im späteren Verlauf der Angriffswelle wurden dann deutlich weniger Angriffe erkannt. Die Rate sinkt hier auf ca. 20% ab.

Invocation Order

Bei der Evaluation des Invocation Order Models (Abbildung 96 auf Seite 199) zeigen sich einige Probleme. Am gravieresten sind die folgenden beiden Probleme identifiziert worden:

1. Da die Bewertung des Modells (= der Anomaly Score) auf der Häufigkeit der einzelnen Übergänge im nichtdeterministischen Automaten zwischen den URLs basiert, wird das erste Vorkommen einer jeden Kombination zweier URLs immer als Anomalie erkannt, da sie zuvor noch nie gesehen wurde.
2. So kommt es zu einer fehlerhaften Bewertung, wenn von einer URL aus sehr oft gleichverteilt viele unterschiedliche URLs aufgerufen werden. Damit sinken die Wahrscheinlichkeiten für alle gültigen Übergänge gleichmäßig ab, so dass der Anomaly Score für die jeweiligen Übergänge ansteigt.

Für beide Probleme haben wir mit der Entwicklung von Lösungsansätze begonnen. Eine Evaluierung der Güte dieser Ansätze ist aber noch nicht erfolgt. Zur Lösung des ersten Problems ist es notwendig, die ersten Beispiele nicht so stark zu gewichten wie später gesehene. Hierzu kann die Konfidenz des Modells dynamisch angepasst werden. Dadurch würden die ersten Vorkommen einer URL z.B. mit einem Multiplikator von 0 gewichtet und dadurch nicht als Angriff erkannt, häufiger Vorkommende aber durch Bewertung mit einer Konfidenz nahe 1 stärker gewichtet werden.

Um das zweite Problem, könnte es sinnvoll sein, absolut häufige Übergänge immer mit einem konstanten Anomaly Score von 0 zu bewerten.

Wenn beide Probleme behoben würden, könnte man die extreme Ausprägung der false positives zu Beginn deutlich verringern und damit auch die true positive Rate steigern. Abgesehen von den angesprochenen Problemen, weißt dieses Modell eine hohe true positive Rate auf. Diese liegt auf dem angewendeten Testdatensatz zwischen 60% und 80%; allerdings bei einer sehr konstanten false positive Rate von 20% bis 30%.

Als alleinstehendes Modell eignet sich dieses Modell nicht, da es nur auf eine spezielle Angriffsart ausgerichtet ist. Neben dem Erkennen von annormalen Aufrufreihenfolgen von URLs erkennt das Modell keine anderen Angriffsstrategien.

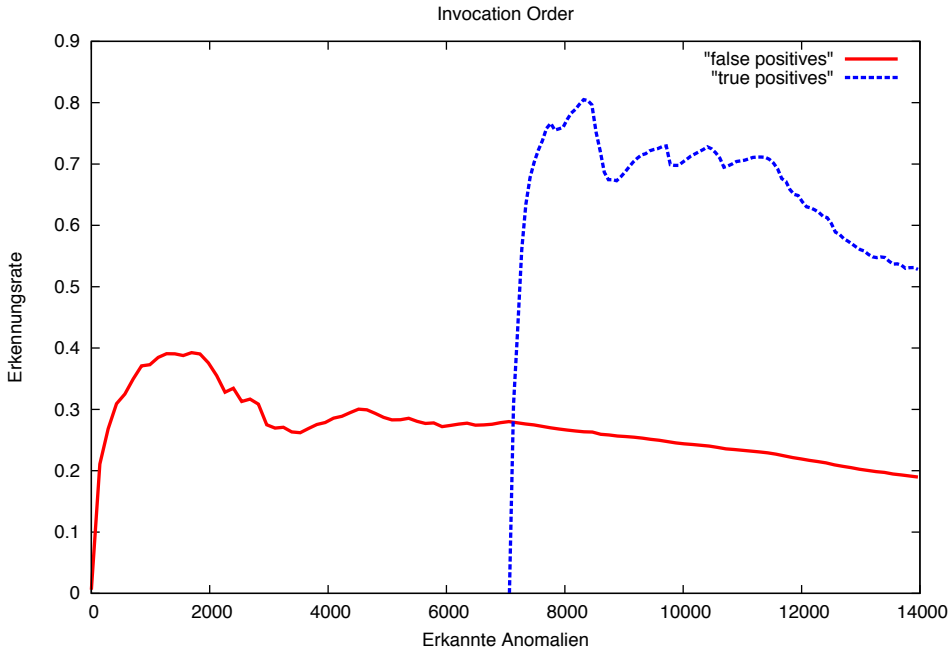


Abbildung 96: Evaluation des Invocation Order Models

7.4 Ausblick

Bei der Realisierung des Web Anomaly Detection Systems hat sich das im Rahmen der Projektgruppe entstandene Framework bewährt. Es bietet durch seine Flexibilität die nötige Plattform, um den multi-model Ansatz aus [30] effizient umzusetzen. Um zu evaluieren wie weit die in [30] beschriebenen Modelle ausreichen, um ein praxistaugliches Intrusion Detection System zu realisieren, wurden die implementierten Modelle einzeln evaluiert. Dadurch kann man die Stärken und Schwächen der einzelnen Modelle besonders gut erkennen. Die Evaluation der aggregierten Modelle ist bis zum jetzigen Zeitpunkt noch nicht erfolgt. Zur Evaluation wurde folgender Ansatz verfolgt. Die Modelle des Frameworks im WADS Aufbau lernen zuerst auf Trainingsdaten. Danach wird mittels Testdaten geprüft, ob Anomalien erkannt werden. Die Trainingsdaten bestehen dabei aus HTTP-Anfragen, die ein typisches „normales“ Surfverhalten widerspiegeln. Die Testdaten dagegen enthalten HTTP-Anfragen, welche in Angriffsszenarien entstanden sind und dementsprechend von dem System auch als Angriffe eingestuft werden sollten.

Der nächste Evaluationsschritt wäre dann, das ganze System live zu testen: Die Modelle werden hier auch als Erstes auf „normales“ Surfverhalten trainiert. Daraufhin wird eine beliebige Webanwendung, deren HTTP-Anfrage über die *AuditConsole* aufgenommen wird, mittels entsprechenden Tools oder auch „per Hand“ attackiert. Das System sollte diese Angriffe nun zeitnah erkennen und entsprechende Warnmeldungen ausgeben.

Da die vollständige Umsetzung dieser Evaluationsansätze bisher noch nicht erfolgte, hat sich die Projektgruppe entschieden das Framework als Open Source Software Projekt (siehe Kapitel 2.7) zur Verfügung zu stellen und so eine Weiterverfolgung der Ansätze zu ermöglichen.

Durch den modularen Aufbau des Frameworks ist es jederzeit möglich zusätzliche Modelle in das Experiment zu integrieren. Die implementierten Modelle decken mit Sicherheit noch nicht den ganzen Raum an Merkmalen ab, an denen sich die vielen verschiedenen Angriffsmuster erkennen lassen. An dieser Stelle, aber auch in der Aggregation der unterschiedlichen Modelle, besteht weiterer Forschungsbedarf.

Index

Literatur

[1] AGGARWAL, C. C., T. J. WATSON, R. CTR, J. HAN, J. WANG und P. S. YU: *A Framework for Clustering Evolving Data Streams*. In: *In VLDB*, S. 81–92, 2003.

[2] ARASU, A. und G. S. MANKU: *Approximate counts and quantiles over sliding windows*. In: *PODS '04: Proceedings of the twenty-third ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, S. 286–296, New York, NY, USA, 2004. ACM.

[3] BALKE, M., T. BECKERS, K. FERNYS, D. HAAK, H. HOMBURG, L. KALABIS, M. KOKOTT, B. KULMANN, K. MÜLLER, C. PRZYLUCZKY, M. SCHULTE, M. SKIRZYNSKI, C. BOCKERMANN und B. SCHOWE: *Zwischenbericht der Projektgruppe 542*. Techn. Ber., Technische Universität Dortmund, 2010.

[4] BERKE, O.: *Book reviews: Introduction to linear regression analysis by D. C. Montgomery, E. A. Peck and G. G. Vining*. Comput. Stat. Data Anal., 39(2):247–247, 2002.

[5] BESAG, J.: *On the statistical analysis of dirty pictures*. Journal of the Royal Statistical Society. Series B, 48(3):259–302, 1986.

[6] BING, L.: *Web Data Mining*. Springer, 2007.

[7] BRÜHL, A. (Hrsg.): *Das V-Modell*. Software - Anwendungsentwicklung - Informationssysteme. Oldenbourg, München [u.a.], 1993.

[8] CAO, F., M. ESTER, W. QIAN und A. ZHOU: *Density-based clustering over an evolving data stream with noise*. In: *In 2006 SIAM Conference on Data Mining*, S. 328–339, 2006.

[9] CARTER, J. L. und M. N. WEGMAN: *Universal classes of hash functions (Extended Abstract)*. In: *STOC '77: Proceedings of the ninth annual ACM symposium on Theory of computing*, S. 106–112, New York, NY, USA, 1977. ACM.

[10] CHARIKAR, M., K. CHEN und M. FARACH-COLTON: *Finding Frequent Items in Data Streams*. In: *ICALP '02: Proceedings of the 29th International Colloquium on Automata, Languages and Programming*, S. 693–703, London, UK, 2002. Springer-Verlag.

[11] CHEN, R., K. SIVAKUMAR und H. KARGUPTA: *An approach to online Bayesian learning from multiple data streams*. In: *In Proceedings of Workshop on Mobile and Distributed Data Mining (PKDD 01)*, S. 31–45, 2001.

[12] CHEN, Y.: *Density-Based Clustering for Real-Time Stream Data*. Techn. Ber., Proc. Of KDD' 07, 2007.

[13] CORMODE, G., F. KORN, S. MUTHUKRISHNAN und D. SRIVASTAVA: *Finding hierarchical heavy hitters in streaming data*. ACM Transactions on Knowledge Discovery from Data (TKDD), 1(4):2, 2008.

[14] CORMODE, G. und S. MUTHUKRISHNAN: *An improved data stream summary: The Count-Min sketch and its applications*. J. Algorithms, 55:29–38, 2004.

[15] DOMINGOS, P. und G. HULTEN: *Mining high-speed data streams*. In: *KDD '00: Proceedings of the sixth ACM SIGKDD international conference on Knowledge discovery and data mining*, S. 71–80, New York, NY, USA, 2000. ACM.

[16] FRIEDMAN, N. und M. GOLDSZMIDT: *Sequential Update of Bayesian Network Structure*. In: *In Proc. 13th Conference on Uncertainty in Artificial Intelligence (UAI 97)*, S. 165–174. Morgan Kaufmann, 1997.

[17] GADATSCH, A. und E. MAYER: *Masterkurs IT-Controlling: Grundlagen und Praxis - IT-Kosten und Leistungsrechnung - Deckungsbeitrags- und Prozesskostenrechnung - Target Costing*. Friedr. Vieweg & Sohn Verlag | GWV Fachverlage GmbH Wiesbaden, Wiesbaden, 3., verbesserte und erweiterte Auflage. Aufl., 2006.

[18] GAROFALAKIS, M., J. GEHRKE und R. RASTOGI: *Querying and mining data streams: you only get one look a tutorial*. In: *SIGMOD '02: Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, S. 635–635, New York, NY, USA, 2002. ACM.

[19] GLOGER, B.: *Scrum: Produkte zuverlässig und schnell entwickeln*. Hanser, München, 2. Aufl., 2009.

[20] GORDEYCHIK, S., J. GROSSMAN, M. KHERA, M. LANTINGA, C. WYSOPAL, C. ENG, S. SHAH, L. LEE, C. MURRAY und D. EVTEEV: *WEB APPLICATION SECURITY STATISTICS 2008*, 2008. [Online; Stand 13. September 2010].

[21] GREENWALD, M. und S. KHANNA: *Space-Efficient Online Computation of Quantile Summaries*. In: *In SIGMOD*, S. 58–66, 2001.

[22] HAN, J.: *Data Mining: Concepts and Techniques*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005.

[23] HAN, J., H. CHENG, D. XIN und X. YAN: *Frequent pattern mining: current status and future directions*. Data Mining and Knowledge Discovery, 15(1):55–86, 2007.

[24] HAN, J., J. PEI und Y. YIN: *Mining frequent patterns without candidate generation*. In: *SIGMOD '00: Proceedings of the 2000 ACM SIGMOD international conference on Management of data*, S. 1–12, New York, NY, USA, 2000. ACM.

[25] HARTUNG, J., B. ELPELT und K.-H. KLÜSENER: *Statistik: Lehr- und Handbuch der angewandten Statistik ; mit zahlreichen, vollständig durchgerechneten Beispielen*. Oldenbourg, München, 14., unwesentlich veränd. Aufl. Aufl., 2005.

[26] HASTIE, T., R. TIBSHIRANI und J. FRIEDMAN: *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Springer series in statistics. Springer, New York, USA, 2001.

[27] HOCHBAUM und SHMOYS: *A best possible heuristic for the k-center problem*. Mathematics of Operations Research, 10(2):180–184, 1985.

[28] IKONOMOVSKA, E. und J. GAMA: *Learning Model Trees from Data Streams*. In: *DS '08: Proceedings of the 11th International Conference on Discovery Science*, S. 52–63, Berlin, Heidelberg, 2008. Springer-Verlag.

[29] KRANEN, P., I. ASSENT, C. BALDAUF und T. SEIDL: *Self-Adaptive Anytime Stream Clustering.*. In: *ICDM*, S. 249–258. IEEE Computer Society, 2009.

[30] KRUEGEL, C., G. VIGNA und W. ROBERTSON: *A multi-model approach to the detection of web-based attacks*. Comput. Netw., 48(5):717–738, 2005.

[31] MANKU, G. S. und R. MOTWANI: *Approximate Frequency Counts over Data Streams*. In: *VLDB*, S. 346–357, 2002.

[32] MASUD, M., J. GAO, L. KHAN, J. HAN und B. THURAISINGHAM: *A Practical Approach to Classify Evolving Data Streams: Training with Limited Amount of Labeled Data*. In: *Data Mining, 2008. ICDM '08. Eighth IEEE International Conference on*, S. 929–934, 2008.

[33] MENDES, L., B. DING und J. HAN: *Stream sequential pattern mining with precise error bounds*. In: *Eighth IEEE International Conference on Data Mining, 2008. ICDM'08*, S. 941–946, 2008.

[34] METWALLY, A., D. AGRAWAL und A. ABBADI: *Efficient Computation of Frequent and Top-k Elements in Data Streams*. In: *Database Theory - ICDT 2005*, Bd. 3363 d. Reihe *Lecture Notes in Computer Science*, S. 398–412. Springer Berlin / Heidelberg, 2005.

[35] O'CALLAGHAN, L., N. MISHRA, A. MEYERSON, S. GUHA und R. MOTWANI: *Streaming-Data Algorithms for High-Quality Clustering*, 2001.

[36] ROSENBLATT, F.: *The Perceptron: A Probabilistic Model for Information Storage and Organization in the Brain*. Psychological Review, 65(6):386–408, 1958.

[37] SHARFMAN, I., A. SCHUSTER und D. KEREN: *A geometric approach to monitoring threshold functions over distributed data streams*. In: CHAUDHURI, S., V. HRISTIDIS und N. POLYZOTIS (Hrsg.): *Proceedings of the ACM SIGMOD International Conference on Management of Data, Chicago, Illinois, USA, June 27-29, 2006*, S. 301–312. ACM, 2006.

[38] TZVETKOV, P., X. YAN und J. HAN: *TSP: Mining top-k closed sequential patterns*. Knowledge and Information Systems, 7(4):438–457, 2005.

[39] WIECZORREK, H. W. und P. MERTENS: *Management von IT-Projekten: von der Planung zur Realisierung ; mit 21 Tabellen*. Xpert.press. Springer, Berlin [u.a.], 2., überarb. und erw. Aufl. Aufl., 2007.

[40] WIKIPEDIA, D. F. E.: *Leetspeak*. http://de.wikipedia.org/wiki/Leetspeak. [Online; Stand 9. Okt. 2010].

[41] WITTEN, I. H. und E. FRANK: *Data Mining: Practical Machine Learning Tools and Techniques*. Hanser, 2. Aufl., 2000.

[42] ZHANG, T., R. RAMAKRISHNAN und M. LIVNY: *BIRCH: an efficient data clustering method for very large databases*. ACM SIGMOD Record, 25(2):103–114, 1996.