Projektgruppe 542
Stream Mining for Intrusion Detection
Zwischenbericht

27. September 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, TU Dortmund

Wintersemester 2009/10

Inhaltsverzeichnis

1 Einleitung

Die zentrale Aufgabe der Projektgruppe 542 ist die Erstellung eines verteilten Netzwerks zur Anwendung und Evaluierung von *stream mining* Algorithmen. Um diesem Netzwerk konkrete Anwendungsfälle zu geben, soll es zunächst zur *intrusion detection* auf verteilten Systemen und zur *spam* Erkennung in Emails eingesetzt werden. Diese Anwendungsfälle dienen dabei lediglich als beispielhafte Anwendungen, anhand derer die Funktionsweise des Netzwerks nachvollzogen, überprüft und getestet werden kann. Das Netzwerk selbst soll so generisch wie möglich aufgebaut werden, um es auch leicht für andere Anwendungen nutzen zu können.

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 gewonne 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 Datenpaket 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

Zur Klärung der Ziele wurde mit den Veranstaltern der Projektgruppe zusammen 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 die Abschnitte

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

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

Darüber hinaus ist in den folgenden Abschnitten die zeitliche Aufteilung auf die beiden Semester nicht weiter festgelegt. Prinzipiell sollte zum Ende des ersten Semesters eine grundlegende Version des Frameworks vorliegen, mit der zwei Anwendungsfälle modelliert und nachgestellt werden.

Für das zweite Semester ist eine Erweiterung zu komplexeren Netzstrukturen vorgesehen.

Vorphase

Innerhalb der Projektgruppe soll ein *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 wäre 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 intelligente Netzwerk von Knoten.

Architektur und Design

Basierend auf den erarbeiteten Anforderungen und Fallbeispielen, soll eine Architektur für ein solches Knotennetz 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 Knotennetzes festlegen zu können. Dazu gehört die Definition verschiedener Knoten und die Festlegung, welche Knoten miteinander verbunden werden sollen (unter der Bedingung, dass die Verbindung der jeweiligen Knotentypen möglich ist). Ein Test auf Kompatibilität zweier zu verbindenden Knoten während der Konfiguration wäre wünschenswert.
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, *streaming* Lernverfahren, soll das Knotennetz die Evaluierung von verschiedenen Lernverfahren ermöglichen. Zur Evaluierung sind Vergleiche der erlernten Modelle oder die Bewertung von Daten mit *label* 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. Hier soll über eine zu entwickelnde Schnittstelle Drittsoftware (z.B. RapidMiner) angebunden werden können.
   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.

Evaluation

Wie bereits erwähnt, sollen die implementierten Stream-Algorithmen hinsichtlich verschiedener Kriterien evaluiert werden, sofern diese auf den Algorithmus sinnvoll anwendbar sind.

Zum einen soll es möglich sein, ähnlich einer Kreuzvalidierung den Vorhersagefehler abzuschätzen. Dies soll auf mehreren Datensätzen mit label durchgeführt werden. Zum anderen sollen die Algorithmen untereinander und mit Batch-Algorithmen verglichen werden in Bezug auf:

1. Güte des Modells
2. Zugesicherte Fehlerschranke
3. Speicherverbrauch

4. Benötigte Verarbeitungszeit pro Beispiel
5. Benötigte Kommunikation zwischen Knoten

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. Protokollierung der Gruppensitzungen (wünschenswert)
2. Beschreibung von Fallbeispielen
3. Ausführliche Beschreibung der Ergebnisse der Anforderungsanalyse
4. Darstellung der entwickelten Framework-Architektur
5. Dokumentation der Programmierschnittstellen (API)
6. Handbuch zur Benutzung des Frameworks.
7. Beschreibung und Evaluierung der implementierten Stream-Algorithmen
8. 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 zusätzliche 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 Implementation
- Implementierung reaktiver Komponenten

Im folgenden werden diese möglichen Erweiterungen genauer spezifiziert.

Validierung und Testen

Es wäre wünschenswert, für relevante Komponenten des Framework 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 vorliegenden Kapitel 1 die Ausgangslage sowie die Aufgabenstellung für unsere Projektgruppe bereits näher erläutert haben, beschäftigen sich die folgenden Kapitel nun mit den von uns erarbeiteten Ergebnissen.

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 auf der Seminarfahrt nach Bommerholz am Anfang des Wintersemesters erarbeitet haben. Die in diesem Kapitel angesprochenen Themen stellen die Grundlage für die Entwicklung von unserem Framework dar.

Im Folgenden gehen wir dann näher auf das von uns entwickelte Framework ein. Wir erläutern in Kapitel 4 das Design des Frameworks und zeigen anhand zweier *use cases* im folgenden Kapitel 5 warum eine solche Architektur Sinn macht um konkrete Anwendungsfälle zu behandeln.
Das Klassendiagramm in Kapitel 6 ist eine gute Möglichkeit um einen Überblick über das implementierte Framework zu bekommen.

Detaillierte theoretische Grundlagen über einzelne Lernverfahren sowohl im Bereich des Data-Mining als auch im Bereich des Stream-Minings werden im Abschnitt 7 dokumentiert.

Die letzten Kapitel geben dann einen Überblick über den Stand der Realisierung (Kapitel **??**) und einen Ausblick auf die geplanten Erweiterungen, die im zweiten Semester umgesetzt werden sollen.

2 Projektkonventionen

2.1 Gruppentreffen

Jede Woche treffen sich alle Projektgruppenmitglieder zu zwei Sitzungen im großen Plenum. Die Treffen finden jeweils Dienstags und Freitags statt und 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 werden 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 Fage, 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.

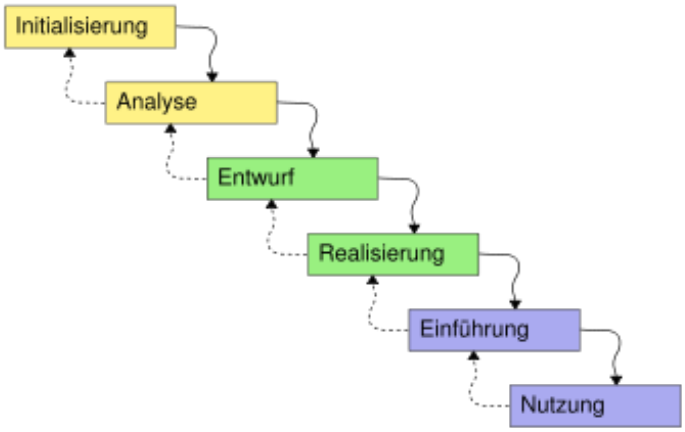


Abbildung 1: *Verfahrensmodell: Wasserfallmodell [13]*

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.

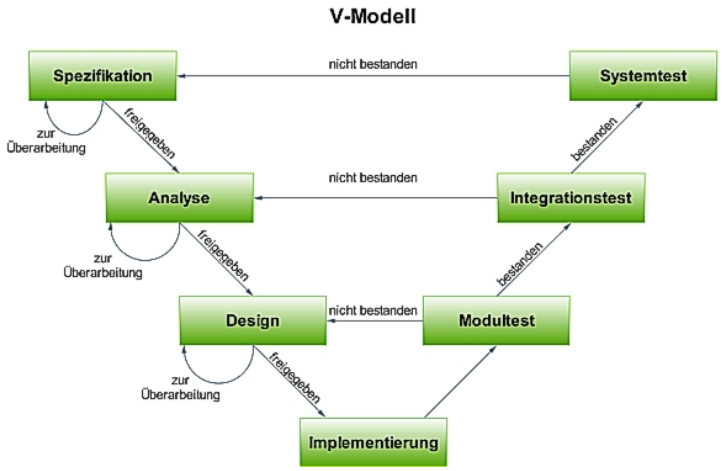


Abbildung 2: *Verfahrensmodell: V- Modell [8]*

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. 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

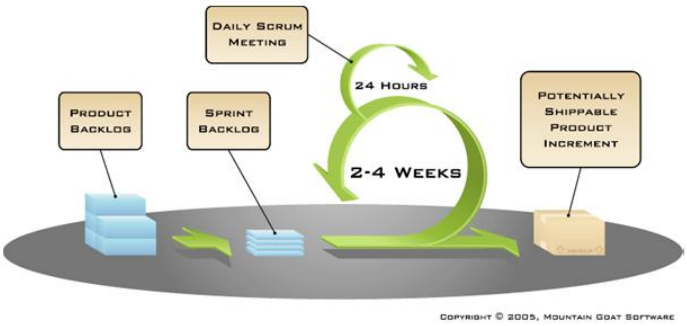


Abbildung 3: *Verfahrensmodell: Scrum Modell [15]*

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.

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.

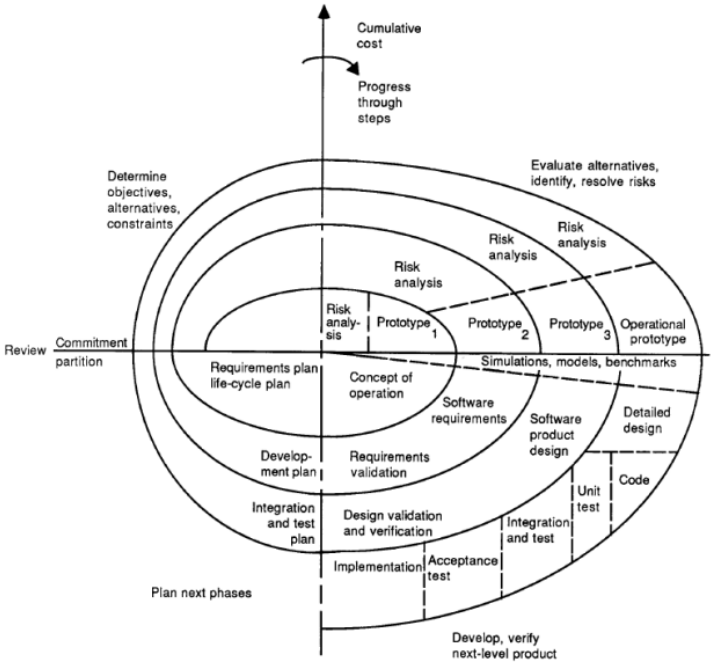


Abbildung 4: *Verfahrensmodell: Inkrementelles Prototyping [24]*

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

In Kaptiel 1.2 wurden bereits die Ziele der Projektgruppe 542 ausführlich definiert und erläutert. Die angegebenen Ziele gilt es bis zum Ende der Projektgruppe im Herbst 2010 zu erreichen. Aus Sicht der Teilnehmer erschien es sinnvoll den grundsätzlichen Aufbau des Frameworks und die Realisierung zweier Anwendungsfälle bereits im ersten Halbjahr der Projektlaufzeit umzusetzen. Die Erstellung weiterer Algorithmen und die Evaluierung der Lernknoten werden die Kernziele des zweiten Halbjahrs sein.
Zusätzlich zu diesen inhaltsbasierten Zielen, ist die ausführliche Dokumentation der Vorgehensweise und Ergebnisse selbstverständlich im jeweiligen Projektzeitraum zu erledigen. Für das erste Halbjahr, das Wintersemester 2009/2010 ergibt sich daher der folgende Terminplan:

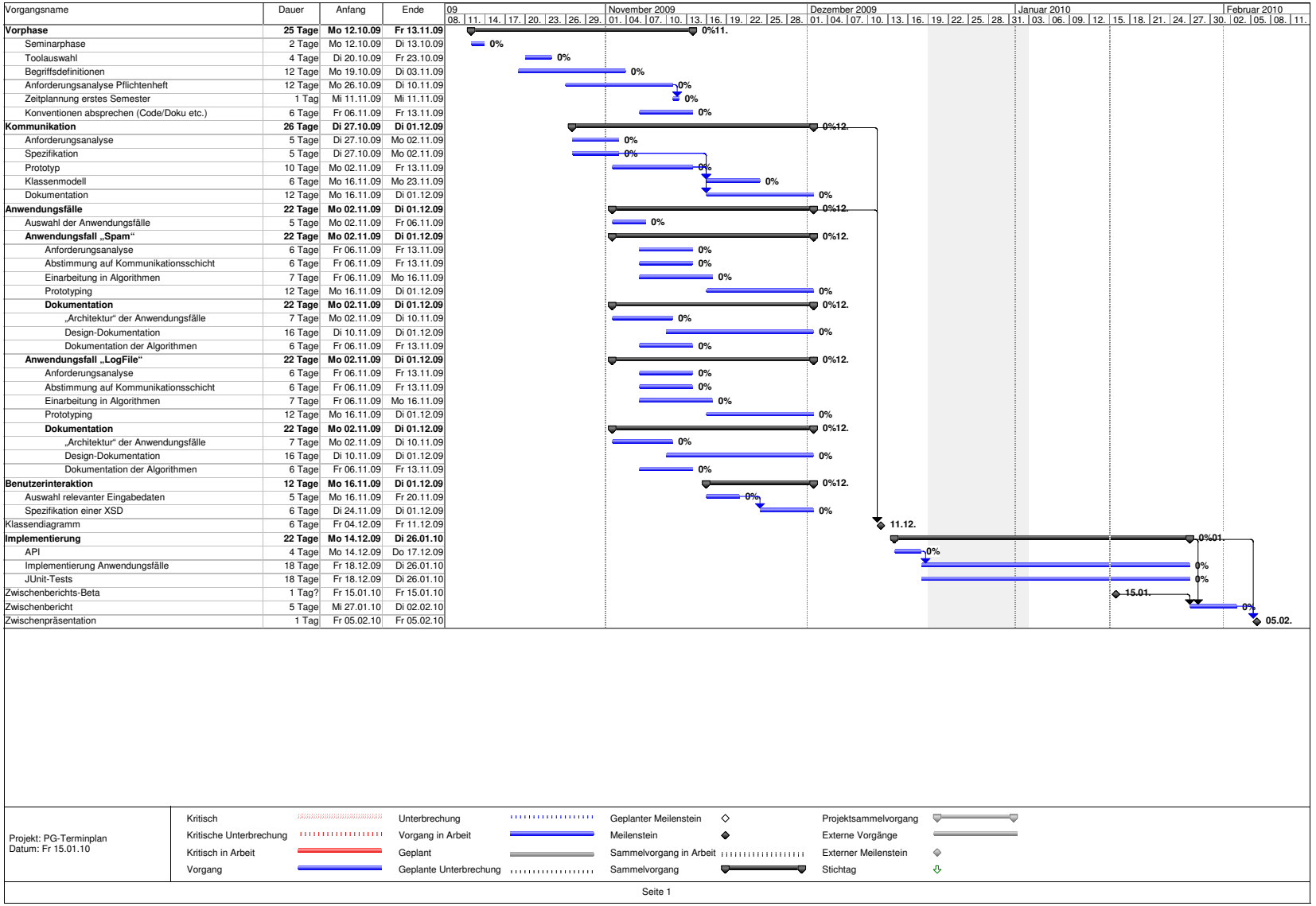


Abbildung 5: *Terminplan für das Wintersemester 2009/2010*

2.5 Begriffe

Grundlage für erfolgreiche Zusammenarbeit in einem Team ist gute Kommunikation. Daher ist es notwendig, dass verwendete Begriffe exakt definiert werden. Die Terminologie, welche wir im Folgenden verwenden werden, wollen wir in diesem Kapitel näher erläutern.

2.5.1 Knoten

Jede funktionale Einheit, welche Eingaben erhält, diese verarbeitet und Ausgaben erzeugt, wird als Knoten (*Node*) bezeichnet. Aufgrund ihrer Funktionalität lassen sich die Knoten in verschiedene Klassen unterteilen.

- **Eingabeknoten** (*InputNodes*) realisieren die erste „Phase“ des Stream-Mining Prozesses. Ihre Aufgabe ist es den Datenstrom (*Stream*) einzulesen und für die weitere Verarbeitung aufzubereiten. Das Ergebnis der *input*-Phase, welche aus mehreren input-Knoten bestehen kann, sind Beispiele (*Examples*). Beispiele dienen als Eingabe für die Folgephase. Eingabeknoten können bspw. Datenleser (*Reader*) oder Parser sein.
- **Vorverarbeitungsknoten** (*PreprocessingNodes*) bilden die zweite Phase. Sie dienen der Aufbereitung der Beispiele und liefern nach Phasenabschluss die Eingabe für die Lernphase. Vorverarbeitungsknoten sind bspw. Aggregatoren (*Aggregators*) oder Filter.
- **Lernknoten** (*LearningNodes*) bilden die eigentliche Lernphase des Prozesses. Ein konkreter Lernknoten ist ein Stream-Mining Algorithmus, der aus den eingegebenen Beispielen ein Modell erlernt und dieses zu bestimmten Zeitpunkten (*Online* Algoritmus) oder auch jederzeit (*Anytime* Algoritmus) ausgeben kann.
- **Validierungsknoten** (*ValidationNodes*) dienen zur Evaluierung der in der Lernphase erzeugten Modelle. Beispiele für diese Knotenklasse sind Modellprüfer (*Modelchecker*) und Modellvergleicher (*Modelcomparer*). Beim Modellvergleich dienen Modelle als Vergleichsmodell, die von Batch-Algorithmen aus Drittsoftware (z.B. RapidMiner) erlernt wurden. Bei der Modellprüfung werden zuvor extrahierte, Beispiele mit label durch das gelernte Modell bewertet.
- **Ausgabeknoten** (*OutputNodes*) dienen zur Ausgabe der Ergebnisse an den Nutzer oder eine weiterverarbeitende Instanz. Die Ausgabe kann bspw. über Email-Konnektoren oder *web services* erfolgen.

2.5.2 Container

Ein Container ist die Umgebung für eine Menge von Knoten. Jeder Knoten ist in genau einem Container eingebettet. Der Container stellt Verwaltungsfunktionalität (z.B. Kommunikation) für die enthaltenen Knoten zur Verfügung. Während sich die Menge der Container im Netzwerk auf mehreren Rechner verteilen kann, liegt ein einzelner Container immer auf genau einem Rechner.

2.5.3 Netzwerk

Das Netzwerk (*Network*) besteht aus einer Menge von Containern, welche auf verschiedene Rechner verteilt sein können. Die Kommunikation zwischen den Knoten im Netzwerk erfolgt über die einzelnen Container. Die Unterscheidung zwischen lokaler (containerinterner) und globaler (containerübergreifender) Kommunikation, die durch den Container zur Verfügung gestellt wird, ist für den einzelnen Knoten transparent (s. Kapitel 2.5.6).

2.5.4 Knotenkommunikation

Link Ein Knoten liefert die Ergebnisse seiner Verarbeitung an eine Menge von Folgeknoten (*Followers*). Die Folgeknoten werden vom Benutzer statisch festgelegt und durch den Namensdienst des Containers aufgelöst.

Topicsubscription Ein Knoten kann sich zu einem Thema (Topic) anmelden und zu einem bestimmten Thema eine Nachricht verschicken. Diese Nachricht erhalten alle Knoten, die sich zu diesem Thema angemeldet haben. Für das Anmelden zu einem Thema und das Verschicken einer Nachricht ist der Container zuständig.

2.5.5 Datenaustausch-Objekte

Zwischen den einzelnen Knoten werden Rohdaten, Beispiele oder Modelle ausgetauscht.

- Als **Rohdaten** (*Raw Data*) werden dabei die unbearbeiteten Daten der Streaming-Quellen bezeichnet. Im Anwendungsfall *Intrusion Detection* könnten dies beispielsweise die Zeilen eines LogFiles sein.
- Ein **Beispiel** (*Example*) ist ein Vektor von Attributen. Es wird durch Eingabeknoten aus den Rohdaten erzeugt, innerhalb der Vorverarbeitungsphase modifiziert und dient als Eingabe für Lernknoten.

- Ein **Modell** (*Model*) ist das Ergebnis von Lernknoten. Modelle können in verschiedenen Variationen auftreten. Beispiele für gelernte Modelle sind **Entscheidungsbäume** (*decision trees*) oder **Klassifikationsregeln** (*classification rules*).

2.5.6 Namensdienst

Jeder Container enthält einen lokalen Namensdienst und evtl. eine Referenz zu einem anderen, entfernten Namensdienst. Dieser hat wiederum einen lokalen Namensdienst und evtl. eine Referenz zu einem weiteren entfernten Namensdienst. Die Namensdienste erstellen dabei eine Kaskadierung, mit einem einzigen globalen Namensdienst, der von allen Namensdiensten direkt oder indirekt erreichbar ist. Will der Container einen Knoten hinzufügen, dann wird der Knoten zunächst lokal gespeichert und danach an den nächsten Namensdienst weitergereicht, der den Knoten speichert und an den nächsten Namensdienst weiterreicht. Um Knoten zu finden, versucht der Container zunächst die Referenz lokal aufzulösen und fragt bei Bedarf seinen übergeordneten Knoten. Dies passiert so lange, bis ein Namensdienst die Referenz auflösen kann oder der letze sie nicht findet.

2.6 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.

2.7 Verwendete Tools

2.7.1 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 2.7.3) - oder auch *build management tools* (siehe 2.7.5). Beides kann in Eclipse durch Plugins integriert werden, so dass all diese Werkzeuge in einer Umgebung zusammenarbeiten können.

2.7.2 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.

2.7.3 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 *Sub-*

clipse heißt. Damit ist es möglich das Projekt bzw. das repository in welchem das Projekt gespeichert ist, direkt aus Eclipse zu verwalten.

2.7.4 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.

2.7.5 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.

2.8 Lizensierung der Ergebnisse

Die Lizenz, unter der die Ergebnisse dieser Projektgruppe veröffentlicht werden, steht bislang noch nicht fest. Sie wird daher erst im Endbericht definiert.

3 Seminarphase

Zum Auftakt unserer Projektgruppe verbrachten wir am Anfang des Semesters zwei Tage im Universitätskolleg Bommerholz in Witten (http://www.tu-dortmund.de/bomholz/). Innerhalb eines festen Zeitplans (siehe S.21) wurde die Herberge genutzt, um sich als Team kennenzulernen und sich in ersten Ansätzen mit der Problemstellung der Projektgruppe auseinanderzusetzen. In den zwei Tage stellte jedes Teammitglied im Rahmen eines Vortrags den anderen Teilnehmern der Projektgruppe mindestens ein Thema vor. Der Vortrag befasste sich in der Regel mit stream mining (Tag 1 S.21) oder intrusion detection (Tag 2 S.22), allerdings wurden auch andere, die Projektgruppe betreffende Themen, wie z.B. Projektmanagment oder Entwicklungstools, präsentiert. Aufgrund der Präsentationen, entwickelten sich durch die intensive Auseinandersetzung der Vortragenden Experten für die jeweiligen Themenbegebiete. Wettbewerbe mit eher weniger Bezug zur Informatik schlossen an die Vortragsphase des ersten Tages an. Dazu gehörten der Bau einer möglichst stabilen Brücke aus einer festgelegten Anzahl von DIN-A4 Blättern mit stark eingeschränkten Hilfsmitteln und Brainstorming zu visionären Technologien, die in 5 bzw. 50 Jahren aktuell sein könnten. Mit diesen Wettbewerben wurde auch der Abend in der „Jägerstube“ eingeleitet, wo u.a. erste Ideen für das Projekt zusammengetragen, Vorstellungen über den Ablauf der Projektgruppe ausgetauscht und ein erster Versuch gemeinsame Termine während des Semesters zu finden unternommen wurden. Zum Abschluss der Seminarphase wurde nach Ende der Vortragsreihe des zweiten Tages der Aufenthalt in Bommerholz reflektiert und über die Organisation der Projektgruppe diskutiert wobei u.a. zwei Projektleiter gewählt wurden.

Zeitplan Bommerholz

Montag - stream mining

Im folgenden wird auf die Inhalte der einzelnen Vorträge näher eingegangen.

9.30 Ankunft

9.45 Begrüßung

10.00 Top-k Algorithmen und Very-Fast Decision Trees (3.1.1) (Tobias)

10.45 Frequent Itemsets in Streams, Hierarchical Heavy Hitters (7.1.2) (Marcin)

11.45 Clustering/Outlier Detection in Streams (3.1.2) (Benedikt)

14.00 Inkrementelle Modelle (3.1.4) (Helge)

15.00 Concept Drift (3.1.3) (Klaudia)

16.00 Lernen auf verteilten Streams (3.2.3) (Matthias S.)

17.00 Sensornetzwerke (3.2.2) (Markus)

19.00 Brücken-Bau- & Visionen-Wettbewerb

Dienstag - intrusion detection

8.40 Collaborative Filtering/Ensemble Methods (3.1.5) (Lukas)

9.30 Überblick über IDS (3.3.1) (Matthias B.)

10.30 Data-Mining und IDS (3.3.3) (Kilian)

11.30 Web-Security Overview (3.3.4) (Daniel)

13.15 Verteilte Systeme mit Java (3.2.1) (Carsten)

14.00 Tools (Marcin)

15.00 Projektmanagement (Klaudia, Kilian)

15.45 Reflektion, Plan, Wahl eine Teamleiters etc.

3.1 Stream - Mining

3.1.1 Top-k Algorithmen und Very-Fast Decision Trees

Die Aufgabe von Top-k-Algorithmen ist die Entdeckung der k häufigsten Elemente aus streams. Da sich der Vortrag auf *very fast decision trees* konzentrierte, sei an dieser Stelle exemplarisch nur auf den Top-k-Algorithmus *lossy counting* (siehe 7.1.2) hingewiesen.

Nach einer kurzen Übersicht zu Baumlernern allgemein und Gütekriterien (im speziellen *information gain - IG*) wurden die Unterschiede zwischen Stream- und konventionellen Batch Baumlernern diskutiert. Wie bei allen Stream-Lernverfahren im Vergleich zu Batch-Lernern, tritt das Problem der unvollständigen Beispielmenge auf: Es kann nur auf den bisher gesehenen Beispielen gelernt werden.

Very fast decision trees (VFDT)

Eigenschaften: VFDT [12] legt, um dieses Problem zu umgehen, die Annahme zugrunde, dass die Verteilung der Elemente des streams über die Zeit konstant bleibt. VFDT ist ein inkrementelles Lernverfahren. Es erlaubt zu jedem Zeitpunkt den Zugriff und das sinnvolle Nutzen des bisher gelernten Modells (*anytime algorithm*). Durch die hohe Verarbeitungsgeschwindigkeit pro Beispiel wird beim Lernen praktisch kein Beispiel verworfen und somit die maximale Informationsmenge des streams genutzt. Nur das Modell und die notwendigsten, aus den Beispielen extrahierten Daten, werden temporär gespeichert - nicht sämtliche Beispiele, was einen geringen Speicherbedarf zur Folge hat.
Das Grundkonzept von VFDT ist wie die meisten Baumlerner leicht zu verstehen und lässt sich daher als exemplarischer Stream-Baumlerner gut in das Gesamtprojekt einbetten.

Vorgehen: VFDT basiert auf Hoeffding trees. Es baut den Baum sukzessive von der Wurzel auf - der Baum wächst also an den Blättern. Sobald genügend Beispiele (für ein definiertes Qualitätskriterium) in einem Blatt vorhanden sind, wird anhand eines Gütemaßes das Attribut bestimmt, das den größten Nutzen bringt. Das Blatt wird durch einen inneren Knoten ersetzt, der auf dieses Attribut testet und so die Beispielmenge weiter in neu erzeugte Blätter aufteilt. In den neuen Blättern wird analog fortgefahren.

3.1.2 Clustering und Outlier Detection in Streams

Clustering bezeichnet das Auffinden von Gruppierungen ähnlicher Elemente in einer Menge von Datenpunkten. Bei der *outlier detection* hingegen wird versucht, Datenpunkte zu identifizieren, die von ebensolchen Gruppierungen abweichen. Intuitiv können outlier als Ausreißer oder Inkonsistenzen in einer Menge von Datenpunkten betrachtet werden.

Sowohl clustering als auch outlier detection gestalten sich auf Datenströmen als schwierig, da aufgrund der Speicherentwicklung von Datenströmen keine ganzheitliche Betrachtung der Datenpunkte möglich ist. Stattdessen wird häufig auf statistischen Daten oder ausgewählten Datenpunkten gearbeitet. Mit der Zeit wurden verschiedene Verfahren entwickelt, um damit zumindest approximativ gute clusterings und outlier detections auf Datenströmen durchzuführen.

Clustering **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 · #chunks* viele Clusterzetren, auf denen ein *high level clustering* durchgeführt werden kann.
CluStream 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.
HPStream ist eine Modifikation von *CluStream*, qualifiziert sich durch Techniken zur Reduzierung der Dimensionalität von Daten jedoch besonders für hochdimensionale Daten.
IncrementalDBSCAN bildet Cluster aufgrund der Dichte der Datenpunkte in der Nachbarschaft eines Datenpunktes.
RepStream arbeitet mit zwei Graphen, in denen Verknüpfungen (1) und die Dichte (2) 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.

Outlier detection Der **MOD Algorithmus** geht von einer Systemstruktur aus, in der es einen *leader node* und m *child nodes* gibt. Child nodes finden nur lokale outlier, während der leader node ein Gesamtbild zusammensetzt. Es wird sowohl dichtebasierte, als auch distanzbasierte outlier detection ermöglicht.
In einem anderen Verfahren wird für jeden eingehenden Datenpunkt ein ***l**ocal **o**utlier **f**actor (LOF)*, basierend auf der Dichte seiner Nachbarschaft, berechnet. Datenpunkte mit einem hohen LOF werden als outlier markiert.
Ferner markieren einige clustering Algorithmen „nebenbei“outlier.

3.1.3 Concept Drift

Dieser Vortrag beschäftigte sich mit der Anomalie-Erkennung bei streaming Daten. Im Maschinellen Lernen wird unter Drift eine feste Variable verstanden, die sich durch Umwelteinflüsse im Laufe der Zeit verändert. *Concept* beschreibt die betrachtete Menge an Daten. Einen Erklärungsansatz für *concept drift* liefert eine Definition aus dem Bergbau, die besagt: „Ein Ausreißer ist eine Beobachtung (oder Teilmenge

von Beobachtungen), welche unvereinbar mit dem Rest der Menge an Daten ist“.

Die Anomalieerkennung wird beispielsweise verwendet, um Emails nach Spam oder Nicht-Spam zu klassifizieren, um so den Nutzer nur relevante Informationen zukommen zu lassen. Ein anderes Beispiel, bei dem die Erkennung von Veränderungen relevant ist, ist die Berechnung des Wareneinsatzes bei Online-Shops. Eine Anomalie tritt auf, wenn der berechnete Wert deutlich unter oder über dem realen Wert liegt. Dabei spielen unterschiedliche Faktoren, wie die Jahreszeit, die wirtschaftlichen Gegebenheiten und Aktionen, eine wichtige Rolle. Um diese Änderungen erkennen zu können, müssen zunächst die Datenmengen beobachtet werden, um anschließend ein Muster erstellen zu können. Die Veränderungen sind erkennbar anhand von extremen Schwankungen in der *noise*-Darstellung, einen *blip* in der Datenmenge sowie einer abrupten oder kontinuierlichen, schrittweisen, positiven Veränderung.

Bei der Erkennung der Veränderungen kommt die Klassifikation von streaming Daten zum Einsatz. Hierbei werden die Daten neu klassifiziert. Daraufhin wird das alte mit dem neuen Label verglichen und die Fehler werden an den Nutzer weitergeleitet. Ein Verfahren bei der Klassifikation ist der ***n**earest **m**ean **c**lassifier (NMC)*. Bei der Beobachtung von Daten bietet die Signalüberwachung eine Unterstützung zur Erkennung von Veränderungen.
Bei der Analyse von Mails werden naive Bayes-Klassifikatoren verwendet, die zunächst die Emails klassifiziert und anschließend eine Textanalyse auf dem Email-Text ausführt. Dabei wird die Häufigkeit der einzelnen Wörter erstellt und mit den Spam-Wörtern verglichen. Weitere Erläuterungen zur Spam-Erkennung folgen im Kapitel Anwendungsfall Spam-Erkennung (siehe 5.2).

3.1.4 Inkrementelle Modelle

Der Vortrag Inkrementelle Modelle behandelt einen Ansatz [10], wie es gelingen kann, ein SVM-Modell sukzessive, Vektor für Vektor, aufzubauen. Es handelt sich dabei um eine echte Online-Methode, da nach der Initialisierung (und einigen anfänglichen Iterationen) zu jeder Zeit ein nutzbares Modell zur Verfügung stehen muss. Die Methode soll des Weiteren möglichst exakt sein. Es existieren beispielsweise „Online“-Varianten, welche vorsehen, eine SVM auf Teildaten zu trainieren und anschließend nichts als die Stützvektoren zu behalten, um auf neuen Trainingsdaten fortzufahren. Solche Lösungen liefern für gewöhnlich keine genauen Ergebnisse.

Der Kern des betrachteten Verfahrens besteht darin, eine Reihe von Bedingungen auf allen bisher analysierten Daten einzuhalten. Dies sind im wesentlichen die ***K**arush-**K**uhn-**T**ucker-Bedingungen (KKT-Bedingungen)*. Die notwendigen Berechnungen erfolgen rein analytisch auf Basis der Trainigselemente und sind vollständig reversibel.

In Abhängigkeit dieser Werte wird das korrespondierende Trainingsmuster zu einer von drei Partitionen hinzugefügt. Sinn dieser Aufteilung ist zu entscheiden, welche der bis dato betrachteten Elemente als mögliche Stützvektoren des aktuellen Modells in Frage kommen könnten. Die drei Partitionen sind:

Die R-Partition: Dies ist die Menge derjenigen Vektoren, welche weder auf noch innerhalb der betrachteten *margin* liegen und so zur Beschreibung der trennenden Hyperebene zunächst nicht beitragen. Diese Menge ist von Bedeutung, da sich die Zugehörigkeit einzelner Trainingsmuster im Verlauf der Modellentwicklung durch Hinzugabe weiterer Traingsmuster ändern kann.

Die S-Partition: Liegt der korrespondierende Vektor direkt auf der margin des momentanen Modells, wird er als Stützvektor der Partition S zugeteilt.

Die E-Partition: Ein Trainigsmuster wird der Partition E als fehlerhafter Stützvektor zugeordnet, wenn er jenseits der margin angeordnet ist. Trainigsmuster aus E müssen jedoch nicht unbedingt falsch klassifiziert werden.

Da das Ziel, den inkrementellen Aufbau des Modells zu ermöglichen, nur erreicht werden kann, wenn in jedem Schritt die KKT-Bedingungen eingehalten werden, muss die Zugehörigkeit der Trainingsvektoren zu den einzelnen Partitionen stets überprüft werden. Dazu ist etwas Buchhaltung zu betreiben. Es müssen einige wichtige Koeffizienten und Kenngrößen aller bisherigen Trainingsinstanzen in Abhängigkeit eines neu hinzugefügten Elements neu berechnet werden. All dies gelingt auf einfache Weise durch Anwendung einiger aus den KKT-Bedingungen hergeleiteten Rechenanweisungen.
Ferner besteht die Möglichkeit, einzelne Lernschritte rückgängig zu machen, was auf natürliche Weise eine effiziente Implementierung des *leave one out* Verfahrens ermöglicht. Statt wie bisher möglicherweise aufwendige Lernschritte durchzuführen, genügt es, das entsprechende Trainingsmuster aus dem Modell zu entfernen.

3.1.5 Collaborative Filtering/Ensemble Methods

Beim kollaborativen Filtern handelt es sich um eine Adaptionsmethode, welche heutzutage auf vielen Online-Plattformen zum Einsatz kommt. Das wohl prominenteste Beispiel aus der Online-Community ist hierbei der Online-Shop *Amazon*, der allein in Deutschland mit über 16 Millionen Kunden zu dem größten und erfolgreichsten seiner Art zählt. Die Idee des kollaborativen Filtern ist recht simpel. Man geht dabei von der Annahme aus, dass Kunden die in der Vergangenheit ein ähnliches Käuferinteresse gezeigt haben, dies auch in Zukunft machen werden.

Neben dem kollaborativen gibt es noch den wissensbasierten sowie den inhaltsbezogenen Ansatz. Des Weiteren sind heutzutage Mischformen, sog. Hybride, sehr verbreitet, da jeder der genannten Ansätze seine Vor- und Nachteile aufweist. Durch geschicktes kombinieren wird versucht, so viele Vorteile wie möglich in seinem System zu vereinigen.
Einen weiteren Unterschied stellt die Repräsentation der Datenbasis dar, auf der dann später die Berechnungen stattfinden. Hier unterscheidet man zwischen dem speicherbasierten sowie dem modellbasierten Ansatz. Im ersten Fall werden jedes Mal alle Datensätze zum Berechnen des neuen Wertes mit einbezogen, während man beim Zweiten nur auf einem vorher erstellten Modell arbeitet. Es ist offensichtlich, dass in der Praxis, wo es oft mehrere Millionen Benutzer oder Einträge in der Datenbank gibt, nur der modellbasierte Ansatz praktikabel ist. Jedoch kann es durchaus sinnvoll sein, sich bei kleineren Systemen für den speicherbasieren Ansatz zu entscheiden.
Die oben erwähnte „Ähnlichkeit“ kann auf unterschiedlichste Arten und Weisen berechnet werden. Ein simples Verfahren ist hierbei das sog. *user-based nearest neighbor* Verfahren, dass von einer Matrix ausgeht, in der die Zeilen die Benutzer und die Spalten die Artikel darstellen. In den jeweiligen Zellen stehen die bereits abgegebenen Bewertungen der jeweiligen Nutzer für Produkte, die sie erworben haben. Hat einer dieser Nutzer eines der Produkte noch nicht erworben und möchte das System eine Empfehlung für dieses vorschlagen, so errechnet es mit Pearsons Korrelationskoeffiezenten die Ähnlichkeit der Nutzer zueinander. Der Benutzer mit dem höchsten Wert ist hierbei demjenigen, für den die Empfehlung abgegeben werden soll, am ähnlichsten. Die Bewertung des ähnlichsten Benutzers wird als Empfehlung für den gesuchten Artikel des aktuellen Benutzer verwendet. Da dieses Gebiet selbst eine Mischform aus den Bereichen *information retrieval*, *information filtering* sowie dem Maschinellen Lernen darstellt, gibt es mittlerweile eine Vielzahl verschiedenster Ansätze, wie Ähnlichkeiten und andere Probleme, wie z.B. das Gewinnen der Informationen (Interessen) der einzelnen Mitglieder, gelöst werden können. Kollaboratives Filtern findet in Internet-Gemeinschaften und auf Internet-Plattformen Anwendung, wenn auf den jeweiligen Benutzer zugeschnittene Informationen aus einer Vielzahl an Daten ermittelt und dem Benutzer präsentiert werden sollen.

Bei den *ensemble methods* geht es wiederrum um das Klassifizieren von Daten bzw. Objekten. Auch in diesem Bereich gibt es mittlerweile eine Vielzahl verschiedenster Lernalgorithmen, deren Ziel es ist, diese Aufgaben zu erledigen. Die Hauptidee der ensemble methods besteht darin, dass man verschiedene Klassifikatoren, welche die gleiche Aufgabe haben, miteinander kombiniert, um damit ein besseres Gesamtergebniss zu erzielen.
Prinzipiell kann man zwischen sequenziell arbeitenden sowie parallelverarbeitenden Klassifikatoren unterscheiden. Bei Erstgenannten gibt es eine Interaktion zwischen den einzelnen Läufen und es ist möglich den Nutzen aus dem vorherigen Lauf in die nächste Iteration mit einzubeziehen. Dies ist vergleichbar mit einer *pipe* auf Unix/Linux Systemen. Ein bekanntes Verfahren hierbei ist das sog. *boosting*. Beim parallelen

Verarbeiten wird die originale Datenmenge am Anfang in mehrere Teilmengen unterteilt. Auf diesen werden dann mehrere Klassifikatoren parallel angewendet. Das sog. *bagging* ist ein Verfahren, welches nach diesem Prinzip arbeitet.

3.2 Verteilte Systeme, Algorithmen und Data Mining

3.2.1 Verteilte Systeme mit Java

In diesem Vortrag wurden verschiedene Mechanismen gezeigt, die dazu dienen verteilte System in Java zu realisieren. Als erstes Konzept wurde *RMI (**r**emote **m**ethod **i**nvocation)* vorgestellt. Dabei werden *Proxy-Objekte* verwendet, die sich nach Aussen wie gewöhnliche Java-Objekte verhalten, intern jedoch die RMI-Kommunikation umsetzten. Diese *proxies* werden von einem Server zur Verfügung gestellt und unter einer eindeutigen ID bei einem Namensdienst registriert. Wenn nun ein Client auf das Objekt zugreifen möchte, stellt er mit der ID beim Namensdienst eine Anfrage und bekommt eine Referenz auf das entsprechende Proxy-Objekt.
Als nächstes wurde XML-RPC vorgestellt. Dabei handelt es sich um eine sehr einfache Form von entfernten Aufrufen. Die Parameter der aufzurufenden Methode werden in eine XML Struktur übertragen, der Aufruf wird gemacht und das Ergebnis wird wieder in XML-Form übertragen. Die Übertragung geschieht hier komplett via HTML. Als mögliche Einträge in der Parameterstruktur erlaubt XML-RPC alle gängigen Basistypen wie Integer, String, Boolean etc., die zu Paketen *(arrays)* zusammengefasst werden können. Ausserdem werden *structs* angeboten, die wie *hash maps* funktionieren. Um die Kodierung in XML muss sich der Entwickler nicht kümmern, da dies die API übernimmt.
Weiterhin wurden *web services* diskutiert. Web services sind nicht zu verwechseln mit Internetanwendungen. In der Regel verwenden Internetanwendungen web services um ihre Aufgabe zu erfüllen. Als Schnittstelle wird hier XML-RPC oder das modernere und mächtigere *SOAP (**s**imple **o**bject **a**ccess **p**rotocol)* benutzt. Ein wichtiger Aspekt im Bezug auf web services ist die Möglichkeit, eine Beschreibung der angebotenen Funktionen zu bekommen, indem man „?WSDL“ anhängt. Die Antwort ist in XML-Form kodiert und neben Methodenamen sind auch die Parameter und deren Typ beschrieben.
Ein weiteres Thema waren *Java message queues*. Dabei wurden implizit auch zwei Modelle der *event-driven* Programmierung vorgestellt. Als Erstes das *queue model* bei dem ein Empfänger von *events* über eine *event queue* verfügt. Diese wird dann nach und nach abgearbeitet, wobei die events von einem *dispatcher* an einen entsprechenden *handler* weitergeleitet werden. Sender können jederzeit neue events einreihen. Wir haben es also mit asynchroner Kommunikation zu tun. Eine bekannte Implementierung dieses Modells findet sich beim *SAX parser (**s**imple **A**PI for **X**ML)*.
Das zweite Modell ist das *publish-subscribe* Modell. Dieses Modell hat Ähnlichkeiten zu einem Newsletter. Events werden nicht an einen Empfänger bzw. eine event

queue, sondern an ein *topic* geschickt. Alle Abonnenten des topics erhalten dann alle events, die an das topic versendet wurden.

3.2.2 Sensor Netzwerke

Der Vortrag „Sensornetzwerke“ war thematisch in fünf Teile gegliedert:

Einleitung Aktuell gewinnen Sensornetzwerke an immer größerer Bedeutung. Verwendung finden sie sowohl in zivilen Bereichen, wie z.B. dem Fahrzeugbau oder der Überwachung von Gebäuden, Habitaten und der Umwelt als auch in militärischen Domänen, wie der intelligenten Kriegsführung. Deshalb wurden die grundlegenden Eigenschaften von Sensornetzwerken sowie ihre speziellen Anforderungen skizziert. Hier wurden auch die wichtigsten Unterschiede zu gewöhnlichen Rechnernetzwerken, wie

- enge Verknüpfung von Energieverbrauch und Lebensdauer,
- stark limitierte Ressourcen,
- unzuverlässiges Kommunikationsmedium, sowie schlecht vorhersagbare Kommunikationsbeziehungen,
- (meist) nicht vorhandener physischer Zugriff auf die Knoten nach dem Ausbringen,

aufgezeigt. In diesem Rahmen wurden einige mögliche Realisierungsansätze speziell für Sensornetzwerke angedeutet.

Angriffszenarien in Sensornetzen Die oben genannten Restriktionen gegenüber gewöhnlichen Rechnernetzwerken lassen bereits erahnen, dass Schutzmechanismen vor Angriffen einen neuen bzw. abgewandelten Ansatz benötigen. Durch potentielle Angreifer ist nicht nur die Integrität der Daten innerhalb des Netzwerks bedroht (z.B. durch die Unzuverlässigkeit des Kommunikationsmediums), sondern auch das Netzwerk selbst. Da in vielen Fällen die Energie eines Sensorknotens endlich ist, können Knoten durch Angriffe nicht nur gestört (z.B. *jamming*), sondern auch regelrecht „zerstört“ werden (z.B. *denial of sleep* oder Routing-Angriffe). Ansonsten wurden in diesem Teil des Vortrags auch kurz „gewöhnliche“ Angriffe, wie z.B. das *spoofing* oder *sniffing* innerhalb eines Netzwerks, erläutert.

Netzwerktechnik Da gerade das spoofing und sniffing in Hinblick auf das *„hacker camp“* von Interesse war, wurde während des Vortrags ein kurzer Exkurs in die Grundlagen der Netzwerktechnik gemacht. Es wurde u.a. erläutert, wie die Sichtbarkeit von Netzwerkpaketen von der Nutzung von *switches* und *hubs* abhängt.

Sniffing Nachdem das Wissen über Kollisionsdomänen und *ARP (**a**ddress **r**esolution **p**rotocol)* aufgefrischt wurde, konnte mit dem Belauschen von Netzwerkverkehr fortgefahren werden. Es wurde z.B. erläutert, wie, durch den Überlauf eines ARP cache in einem switch, die Pakete außerhalb der eigenen Kollisionsdomäne abgehört werden können. Zusätzlich wurde auf die Funktionsweise von Netzwerk-TAPs eingegangen.

Intrusion detection in Sensornetzen Der Vortrag wurde abgerundet durch einen Überblick von intrusion detection Systemen für Sensornetzwerke. Es wurde erläutert, dass eine Anomalieerkennung in Sensornetzwerken unverzichtbar ist und durch ein dezentralisiertes System auf Basis von *watchdogs*, welche die Kommunikations im Empfangsradius des eigenen Knoten überwachen, realisiert werden kann.
Ein relativ neuer Ansatz für intrusion detection in Sensornetzwerken ist die Verwendung eines Agentensystems mit dem Namen ***a**nt **c**olony **s**ystem (ACS)*. Auch wenn in den vorliegenden Quellen [6, 20, 7] nur Ansätze genannt wurden, bot das Thema Potential und wurde zum letzten Schwerpunkt des Vortrags. Es handelt sich um ein Agentensystem, das, angelehnt an das Verhalten von Ameisen auf der Futtersuche, indirekte Kommunikation durch Markierungen entlang des Weges betreibt. Da der Ursprung von ACS im Bereich des ***t**raveling **s**alesman **p**roblems (TSP)* liegt, scheint dieser Ansatz für die Entdeckung von jamming und denial of service sowie Routing-Angriffen geeignet.

3.2.3 Lernen auf verteilten Streams

Das stream mining auf verteilten Streams unterscheidet sich vom klassischen stream mining dadurch, dass die Kommunikation zwischen den Knoten im System *overhead* erzeugt, welcher die Performanz stark negativ beeinflusst. Daher mussten Ansätze gefunden werden, die es erlauben, die Kommunikation zwischen den Knoten zu minimieren.

Erste und einfache Ansätze hierfür sind zum Beispiel

- Ungenauigkeiten zu akzeptieren oder
- zentrale Knoten einzurichten, die den Status der Knoten überwachen und so eine Kommunikation aller Knoten untereinander unnötig machen.

So könnte ein System, in welchem die Überschreitung eines bestimmten Grenzwertes global interessant ist (z.B. fehlgeschlagene Anmeldeversuche an gruppierten Webservern), von einem zentralen Knoten überwacht werden. Tritt ein relevantes Ereignis an einem der Knoten ein, schickt dieser eine Meldung an die zentrale Einheit. Die anderen Knoten müssen nicht informiert werden. Zusätzlich könnte man das System so konfigurieren, dass nur jedes fünfte Ereignis gemeldet wird. Die zentrale Einheit

hat dann zwar keinen Überblick über die konkrete Anzahl relevanter Ereignisse im System, sie kann diese aber ungefähr abschätzen. Der Fehler kann in diesem Fall maximal $5 \cdot (AnzahlderKnoten)$ Ereignisse betragen.

Komplexere Ansätze sind oftmals nur für spezielle Problemstellung zu verwenden. Exemplarisch wurden die Ideen der Veröffentlichungen

- „Catching elephants with mice“ [14] und
- „A geometric approach to monitoring threshold functions over distributed data streams“ [21]

vorgestellt. Die Reduktion des overheads der Kommunikation wird durch geschicktes *sensor sampling* [14] und Minimierung unnötiger Meldungen [21] erreicht.
Die Ansätze werden in den Kapiteln 7.2.1 und 7.2.2 detailierter erläutert.

3.3 Intrusion Detection

3.3.1 Überblick über IDS

Wenn man über intrusion detection Systeme (IDS) spricht, sollte man sich zunächst die Frage stellen, was ein IDS ist und wofür man es braucht.
Definition:

> „Ein Intrusion Detection System (IDS) ist ein System zur Erkennung von Angriffen, die gegen ein Computersystem oder Computernetz gerichtet sind.“ [1]

Es handelt sich also um ein System, welches dem Schutz des Computersystems oder eines ganzen Netzwerkes dient. Aber warum muss man sich überhaupt schützen? Welchen Arten von Angriffen sind Computersyteme in der Realität ausgesetzt?

Heutzutage laufen auf einem einzigen Computersystem sehr viele Anwendungen. Da Anwendungen von Menschen programmiert sind, passiert es häufig, dass sich bei der Ausführung ein Fehler einschleicht. Durch diese Fehler sind die Systeme mehr oder weniger kritisch angreifbar. Da die meisten Firmen und privaten Haushalte heute auch mit dem Internet verbunden sind, ergeben sich neue Gefahrenquellen. Doch nicht nur Fehler in Programmen sind kritisch. Die Gefahr kann auch aus dem eigenen Netzwerk stammen. So kommt es immer wieder vor, dass z.B. Accounts und dazu gehörende Benutzerrechte in einem Firmennetz missbraucht werden, da sich Mitarbeiter nicht ausgelogt haben oder ihre Logindaten bekannt geworden sind.

Ist das System durch eine Sicherheitslücke oder das fahrlässige Verhalten eines Nutzers gehackt worden, stehen dem Angreifer häufig alle Daten zur Verfügung. Da es

[1] http://de.wikipedia.org/wiki/Intrusion_Detection_System

sich meistens um sensible Daten handelt ist es wichtig diese zu schützen.
Zur Abwehr in beiden Szenarien kommen Intrusion Detection Systeme zum Einsatz. Im Falle einer Sicherheitslücke im System hilft ein network intrusion detection System (NIDS). Dieses ist ein zentral für das Netzwerk installiertes System, welches den Netzwerkverkehr überwacht und Angriffe erkennen kann. Im Falle von geklauten Log-In-Daten eines Nutzers würde dieses System allerdings keine Unstimmigkeiten entdecken. In diesem Fall müsste man ein host intrusion detection System (HIDS) nutzen. Dieses ist lokal auf jedem Computer installiert und überwacht beispielsweise das Verhalten des Nutzers (siehe Kapitel 3.3.3).

Man kann auch beide Systeme kombiniert einsetzen. Dann spricht man von einem hybrid intrusion detection System (Hybrid IDS).

Jede Variante des IDS nutzt unterschiedliche Erkennungsverfahren. Drei immer wieder genutzte Verfahren sind

- Integritätsprüfung,
- Signaturprüfung und
- Anomalieerkennung.

Die **Integritätsprüfung** wird meist für HIDS verwendet. Sie überwacht die ausführbaren Dateien auf einem Computersystem, indem sie eine Datenbank mit Hashwerten erstellt. Diese wird so gespeichert, dass sie nur gelesen werden kann. Wenn nun eine Datei ausgeführt werden soll, wird der Hashwert neu berechnet und mit dem initial erstellten verglichen. Stimmen beide überein wird das Programm ausgeführt. Tun sie es nicht wird das Ausführen des Programms verhindert.

Die **Signaturprüfung** dagegen wird sowohl in NIDS als auch in HIDS eingesetzt. Der bekannteste Vertreter dieser Prüfmethode ist der Virenscanner (HIDS). Dieses Verfahren sucht nach bekannten „gefährlichen“ Signaturen in Dateien oder Netzwerkströmen. Ein großer Nachteil ist, dass es nur Angriffe erkennen kann, die schon einmal aufgetreten sind und zu denen eine Signatur existiert. Daher ist hier das Aktualisieren der Signaturen essentiell.

Bei der **Anomalieerkennung** versucht man diese Problematik zu vermeiden. Das System erstellt zunächst anhand von mathematischen und statistischen Modellen ein Nutzungsprofil für eine Anwendung. Es werden Schranken für die normale Nutzung und einen Angriff festgelegt. Danach berechnet das System für jedes eingehende event (z.B. ein Netzwerkpaket oder den Aufruf einer Webseite) den sog. anomaly score. Überschreitet dieser die vorher definierten Schranken, so wird z.B. das Netzwerkpaket verworfen oder ein Alarm ausgelöst. Die Rate der fälschlicherweise als ungefährlich erkannten events (false positive) ist zwar enorm niedrig, aber nicht gleich null. Daher ist dieses System zwar nahezu wartungsfrei, aber die Ergebnisse müssen fortlaufend überprüft werden.

Fazit: Ein IDS hilft Angriffe zu erkennen. Man kann sich allerdings nicht zu 100% auf ein einzelnes oder auf eine beliebige Kombination von intrusion detection Systemen verlassen, da keines der System alle Angriffe erkennen kann und manche Systeme eventuell - wenn auch nur selten - fehlerhafte Aussagen machen.

3.3.2 Testen von IDS

Ein IDS ist, wie in Kapitel 3.3.1 gesehen, häufig nur so gut wie seine Signaturen. Aber auch wenn kein Signaturerkennungsverfahren zum Einsatz kommt, kann man ein IDS nicht einfach installieren und es dann unbeachtet lassen.

Da Software nie fehlerfrei ist, bessern die meisten Hersteller ihre Software durch patches und neue Versionen aus. Doch auch in den neuen Versionen schleichen sich häufig wieder Fehler ein. Dadurch gibt es immer wieder neue Arten Anwendungen anzugreifen. Die neuen Angriffstechniken sind meist schon kurz nach dem Erscheinungsdatum der neuen Versionen bekannt und werden direkt ausgenutzt. So entstehen stetig neue Herausforderungen an das benutze IDS.

Um sicher gehen zu können, dass das genutzte IDS diesen neuen Angriffen stand hält, muss man sein eigenes IDS regelmäßig testen. Es gilt Sicherheitslücken zu finden bevor Angreifer diese aufspüren und ausnutzen. Um nicht jeden Angriff einzeln von Hand ausführen zu müssen existieren Tools, die viele Angriffe auf ein IDS starten und die Ergebnisse auswerten können. Doch auch diese Tools kennen nur die bekannten Angriffe. Daher muss das Wissen über neue Angriffsmöglichkeiten wachsen. Da es oft sehr kompliziert und vor allen Dingen sehr zeitaufwändig ist, neue Angriffsmethoden zu entwickeln, kann man auch versuchen von den Angreifern abzuschauen. Genau für dieses Aufgabe wurden sogenannte **honeypots** entwickelt. Es handelt sich dabei um Systeme, die nach außen so tun, als würden sie bestimmte Dienste anbieten und intern alle Informationen über den Angriff und den Angreifer mitschreiben. Das Ganze basiert auf einem einfachen Ansatz: honeypots sind Systeme auf denen keine Dienste für den normalen Benutzer laufen. Zusätzlich ist die Existenz der Honeypots dem normalen Benutzer zumeist unbekannt. Daher wird ein normaler Benutzer diese Computer nie ansprechen oder versuchen, irgendwelche Dienste auf diesen zu nutzen. Kommt es nun zu einer eingehenden Verbindung auf dem honeypot, so kann man davon ausgehen, dass es sich um einen Angriff auf den Computer handelt. Aus den mitgeschriebenen Informationen lassen sich häufig Erkenntnisse über neuartige Angriffe ableiten.
Doch honeypots bergen auch Gefahren. Auf ihnen sind häufig veraltete Versionen einer Software installiert, die die Angreifer anlocken sollen. Dies stellt ein Sicherheitsrisiko dar, da es vorkommen kann, dass der honeypot dem Angreifer den Einstieg ins eigene Netzwerk noch erleichtert. Aus diesem Grund müssen die honeypots besonders gut abgesichert werden.

3.3.3 Data-Mining und IDS

Ein intrusion detection System dient zur Erkennung von Angriffen auf einen einzelnen Rechner oder ein ganzes Rechnernetz. Um einen unautorisierten Eindringling zu erkennen, nutzen intrusion detection Systeme verschiedene Methoden. Die populärsten Methoden versuchen, Eindringlinge anhand gängiger Angriffsstrategien (misuse detection) oder aufgrund von auffäligem und abnormen Verhalten (anomaly detection) zu erkennen. Das Thema dieses Vortrags war die Nutzung von data mining-Techniken für intrusion detection Systeme und beruhte auf dem Artikel „A Data Mining Framework for Building Intrusion Detection Models“ [17].

Um die Nützlichkeit von data mining-Techniken in intrusion detection Systemen zu verstehen, ist zunächst ein Überblick über misuse- und anomaly detection notwendig:

Ein *misuse detection System* versucht Regeln für bekannte Angriffe aufzustellen, um diese so früh wie möglich erkennen zu können. Eine solche Regel könnte beispielsweise lauten:

> „Wird ein Passwort innerhalb von zwei Minuten viermal falsch eingegeben, so liegt ein Angriff vor.“

Um ein wirksames misuse detection System zu erstellen, wird ein hohes Fachwissen über verschiedene Angriffstechniken benötigt, da dieses die Grundlage für die Erstellung der Regeln bildet. Außerdem kann ein solches System immer nur bereits bekannte Angriffe erkennen und nicht auf neue Entwicklungen reagieren (siehe Kapitel 3.3.1). Da die Anzahl der Angriffsstrategien aber sehr schnell wächst, ist eine Aktualisierung der Regelwerke zwingend erforderlich um die Qualität des Systems zu gewährleisten Diese ist jedoch mit einem hohen Wartungsaufwand verbunden. Wünschenswert wäre in diesem Bereich eine automatisierte Regelgenerierung, welche das notwendige Fachwissen und den Wartungsaufwand reduziert und gleichermaßen Flexibilität und die Reaktionsfähigkeit auf neue Angriffe steigert.

anomaly detection Systeme definieren für die Erkennung von Eindringling eine Reihe von Regeln, welche das erwartete Nutzerverhalten beschreiben. Weicht ein Nutzer von diesem Verhalten ab (tritt eine Anomalie auf), so wird von einem potenziellen Angriff ausgegangen. Anwendungsbereiche für diese Art der IDSe liegen vor Allem bei den sogenannten host intrusion detection Systemen (HIDS), die Angriffe „von innen“ erkennen sollen. Beispielsweise ließe sich das Verhalten verschiedener Beschäftigter einer Firma darstellen, um bei Wirtschaftsspionage und Ähnlichem den Angriff von Innen zu verhindern.
Die Nachteile dieser Systeme liegen ebenfalls in dem hohen benötigten Fachwissen (diesmal nicht auf Angriffs- sondern Applikationsebene) und dem hohen Wartungsaufwand. Auch in diesem Fall wäre es sinnvoll, die Regeln für die Definition der Nutzerprofile automatisch generieren zu lassen.

Data minig-Techniken können bei der Erstellung von IDSen von großem Nutzen sein. Für die Erstellung eines misuse detection Systems müssen zunächst Rohdaten (beispielsweise aus Logfiles etc.) gesammelt werden. Können diese Daten klassifiziert werden, so lassen sich die benötigten Regeln durch Regelgeneratoren wie *Ripper*, welche auf DM-Technologien basieren, bilden. Die Erstellung kann also größtenteils automatisiert werden, allerdings werden künstliche, klassifizierbare Testdaten benötigt. Mit einigem Aufwand ließen sich für die Gewinnung dieser Daten aber *Honeypots* (siehe Kapitel 3.3.1) nutzen.
Auch bei den *Anomaly Detection Systemen* können Dataminig-Techniken wie der Apriori für die Regelgenerierung genutzt werden. Durch das Mitloggen der Benutzerinteraktionen können im Nachhinein Analysen auf diesen Daten durchgeführt werden. Anhand der Daten lässt sich dann mit Hilfe des Apriori ein Regelwerk als Nutzerprofil erstellen. Ein Beispiel für ein solches Profil könnte etwa Folgendes sein:

- Programmierer $\Rightarrow$ arbeitet nachmittags
- Programmierer $\Rightarrow$ programmiert und compiliert mit Eclipse
- (Programmierer, Vormittag) $\Rightarrow$ mailt mit Thunderbird.

Fazit: Data mining-Techniken können zur automatisierten Erstellung von Intrusion Detection Systemen genutzt werden. Eine vollständige Automatisierung ist aber nicht möglich. Der Erstellungsprozess kann durch die Verwendung von Data-Mining aber weitreichend verschlankt und vereinfacht werden.

3.3.4 Web-Security Overview

Das folgende Kapitel gibt einen Überblick über das Thema Sicherheit im Web und gliedert sich in die Abschnitte Motivation, OWASP Top 10, Tools, Praxis und Fazit.

Motivation
Nahezu jedes Unternehmen und jede öffentliche Einrichtung nutzt komplexe Netz- und Kommunikationsstrukturen und gibt alleine durch den Anschluss an das Internet ein Teil der Kontrolle über diese ab. Das hieraus unmittelbar Gefahr folgt, wird spätestens dann deutlich, wenn man sich die Statistik des Open Web Application Security Consortiums (OWASP) ansieht, die besagt, dass sich in 96,85% (!) aller Webanwendungen kritische Sicherheitslücken identifizieren lassen [2].
Um sich vor Angriffen schützen zu können, sollte man sich als Erstes vor Augen führen, wer überhaupt angreift. Im Volksmund wird dabei immer von „Hackern“ gesprochen. Dies sollte allerdings deutlich differenzierter betrachtet werden [16]:

- **„Hacker“:**
  Der „Hacker“ im eigentlichen Sinne ist nur ein experimentierfreudiger Programmierer der technisches Interesse zeigt und keine kriminellen oder wirtschaftlichen Absichten verfolgt.

- **„Cracker“:**
  Der „Cracker“ dagegen ist schon eher ein „böser Hacker“, denn er steckt kriminelle Energie in das „Hacking“ und versucht sich rechtwidrige Vorteile zu verschaffen.
- **„Script-Kiddies“:**
  Der Tätertyp „Script-Kiddie“ agiert dabei ohne umfangreiches Hintergrundwissen, und probiert aus Neugier vorgefertigte Angriffstools aus und setzt sie auf willkürliche Ziele an.
- **„Insider“:**
  Ein „Insider“ ist ein „Cracker“ von dem erhöhte Gefahr ausgeht, weil er privilegiertes Wissen über sein Ziel hat und so möglicherweise Kenntnisse über Schwachstellen besitzt. Ein typischer „Insider“ wäre ein frustrierter (ehemaliger) Mitarbeiter.
- **Wirtschaftsspionage:**
  Eine weiterer Hackertyp ist der Wirtschaftsspion, welcher Betriebsgeheimnisse wie z.B. Wissen über Innovationen zu stehlen versucht und die Informationen zum eigenen Vorteil verwenden will.
- **„Hacktivism“:**
  Eine neuere Form eines Tätertypens ist „Hacktivism“, einer Mischung aus „Hacking“ und politischen bzw. sozialem Aktivismus [5].

Was zudem oft in den Hintergrund gerät - aber auch wichtig für das Verständnis der Motivation eines „Hackers“ ist - ist die Tatsache, dass hinter dem „Hacken“ eine gewisse Ethik steckt, die sich auf das Buch „The Hacker Manifiesto“ eines anonymen Autors mit dem Pseudonym „The Mentor“ stützt [3]. Zu den Grundsätzen der „Hacker“-Ethik gehört zum Beispiel, dass das Eindringen in Computersysteme zum Zweck des Vergnügens und der Wissenserweiterung akzeptabel ist, aber damit niemals der Diebstahl oder die Manipulation von Daten oder die Verfolgung kommerzieller Ziele zu rechtfertigen sind [4]. Zusammengefasst sind die oben beschriebenen Tätertypen „Cracker“, „Insider“ und die Wirtschaftsspionage in keinster Weise mit der ursprünglichen „Hacker“-Ethik vereinbar.

OWASP Top 10
Die OWASP Top 10 sind eine Auflistung des Open Web Application Security Projects (`http://www.owasp.org`), die die zehn aktuellsten Probleme im Bezug auf Webanwendungen - und damit typische Angriffsarten und Sicherheitslücken - aufzeigt [1].

1. **Cross Site Scripting (XSS):**
   An erster Stelle der OWASP Top 10 steht *Cross Site Scripting*, was überall möglich ist, wo Daten, die von einem Benutzer eingegeben werden, in ungeprüfter und/oder ungefilterter Form vom Server zurückgesendet und im Webbrowser

ausgegeben werden. *XSS* ermöglicht das Ausführen von beliebigen Skriptcode im Webbrowser eines Opfers.

2. **Injections Flaws:**
   Den zweiten Platz belegen *Injections Flaws*, womit u.a. *SQL-Injections* gemeint sind. Diese sind möglich, wenn eine Applikation gesendete Daten auf Seiten des Servers als Command, Query oder Teil einer solchen interpretiert. Dem Angreifer ist es dadurch möglich beliebige Daten zu lesen, schreiben, erweitern oder auszuführen.

3. **Insecure Remote File Include (RFI):**
   An dritter Stelle steht *Insecure Remote File Include (RFI)*, von dem man spricht, wenn Entwickler der Eingabe eines Users vertrauen und die von ihm gesendeten Daten in einer File-, Include- oder Stream-Funktion ungeprüft und/oder ungefiltert verwenden. *RFI* betrifft in erster Linie die PHP-Befehle `require` und `include` bzw. `require_once` und `include_once`. Der Angreifer kann durch diese Sicherheitslücke externen Code ausführen lassen und ganze Systeme kompromittieren.

4. **Insecure Direct Object Reference:**
   Eine ähnliche Angriffsart stellt der vierte Platz der OWASP Top 10, die *Insecure Direct Object Reference*, dar. Diese liegt vor, falls Objekte, wie z.B. Dateien, Ordner, Datenbankeinträge oder Schlüssel durch die URL oder als Form-Parameter eingebunden werden. Ohne Implementierung eines access control checks können die Parameter von einem Angreifer manipuliert werden, der sich so Zutritt in eigentlich geschützte Bereiche verschafft.

5. **Cross Site Request Forgery (CSRF):**
   Den fünften Platz belegt *Cross Site Request Forgery (CSRF)*, das sich gegen eingeloggte User einer Webanwendung richtet, deren Rechte vom Angreifer ausgenutzt werden. Das Opfer wird mittels technischer Maßnahmen oder zwischenmenschlicher Überredungskunst dazu gebracht, vom Angreifer präparierten Code auszuführen, mit dem sich dieser z.B. einen neuen Useraccount für sich in einem Forum anlegt, was mit seinen Rechten nicht möglich wäre.

6. **Information Leakage & Improper Error Handling:**
   Diese Angriffsart bezeichnet die Ausnutzung von Fehlermeldungen zum Informationsgewinn. Der Angreifer gelangt an Informationen, wie z.B. über interne Abläufe, Zustände oder Konfigurationen einer Webanwendung, in denen er dann Fehler provoziert.

7. **Broken Authentication & Session Management:**
   Broken Authentication & Session Management liegt überall dort vor, wo ein Session Management System fehlerhaft implementiert wurde. Die Fehlerquellen steigen in der Regel mit wachsendem Funktionsumfang, wie z.B. Logout, Geheimfrage oder Erinnerungsmail, und machen es dem Entwickler schwierig keine Fehler zu machen, durch die ein Angreifer User-Sessions übernehmen kann.

8. **Insecure Cryptographic Storage:**
   Auf Platz 8 steht *Insecure Cryptographic Storage*, ein Sicherheitsrisiko, welches überall vorliegt, wo sensitive Daten durch Kryptographie geschützt werden sollen, die entsprechenden Funktionen aber fehlerhaft implementiert oder genutzt werden. Ein Angreifer hat durch diese Sicherheitslücke die Möglichkeit sensitive Daten einzusehen. Der größte Fehler hier ist allerdings gefährliche Daten einfach unverschlüsselt zu lassen.

9. **Insecure Communications:**
   Eine ähnliche Sicherheitslücke stellt *Insecure Communications* dar, die sich allerdings nur auf unverschlüsselte Netzwerke bezieht, die nicht durch z.B. SSL geschützt werden. Es ist dringend notwendig, dass jede Verbindung, die nicht jedem zugänglich seien soll, geschützt wird.

10. **Failure to Restrict URL:**
    Schlussendlich stellt die *Failure to Restrict UR*L-Lücke den letzten Platz der aktuellsten Sicherheitslücken in Webanwendungen dar. Diese liegt überall dort vor, wo bestimmte Bereich nicht durch Login-Mechanismen o.Ä. geschützt werden sollen, sondern durch die Eingabe der korrekten URL. Ein hochmotivierter Angreifer mit etwas Glück errät die richtige URL durch Ausprobieren und verschafft sich Einblick in sensitive Daten.

Tools

Die Masse an Tools, die helfen können Sicherheitslücken zu erkennen, ist immens. Grob gesagt lassen sich aber alle Tools in eine von zwei Kategorien einsortieren: Zum Einen in Tools, die auf eine einzige Sicherheitslücke spezialisiert sind, und zum Anderen in Vulnerability Scanner, die eine Webanwendung automatisch auf eine Reihe von Sicherheitsrisiken untersuchen.
Zwei Beispiele für allgemeine Vulnerability Scanner sind der *Acunetix Web Security Scanner* (http://www.acunetix.com) und das *Metasploit Framework* (http://www.metasploit.com). Der *Acunetix Web Security Scanner* läuft unter Windows, wird kommerziell vertrieben, soll sehr mächtig sein und eine hohe Prüftiefe besitzen. Von der Funktionalität her soll er eine Reihe der OWASP Top 10 Angriffe simulieren können, wie z.B. *XSS* und *SQL-Injections*. Das *Metasploit Framework* ist dagegen ein Open Source Vulnerability Scanner, der auf allen gängigen Plattformen läuft (Windows/Linux/Mac OS X) und ca. 450 Exploits für verschiedene Systeme kennt, die vom Benutzer konfiguriert und angewendet werden können.
Zu den speziellen Tools gehört zum Beispiel *Scrawlr*, der in die Kategorie der SQL-Injections Scanner gehört. Scrawlr wurde von Microsoft und Hewlett Packard (http://www.hp.com) entwickelt, ist kostenlos und passt nach dem Scan einer Webanwendung SQL-Exploits an diese an. Ein weiterer SQL-Injections-Scanner ist *sqlmap* (http://sqlmap.sourceforge.net). Das Tool ist Open Source und unterstützt die Infiltrierung aller gängigen Datenbanken, wie MySQL, Oracle, PostgreSQL oder Microsoft SQL Server. Darüber hinaus gibt es noch eine Reihe von speziellen Tools für andere Sicherheitslücken, wie z.B. mit *Springenwerk Security Scanner* (http://

springenwerk.org) einen XSS Scanner, *WordPress Scanner* (http://blogsecurity.net/wordpress/tools/wp-scanner), einen Scanner für WordPress Blogs oder *OWASP Sprajax* (http://www.owasp.org/index.php/Category:OWASP_Sprajax_Project), einen Blackbox-Scanner für Ajax-Anwendungen.

Praxis
Möglichkeiten, das theoretisch gewonnene Wissen in der Praxis zu vertiefen, bietet zum Beispiel die Google Code University (http://code.google.com/intl/de-DE/edu). Dort findet man Tutorials und Beispielanwendungen für Informatik-Studenten und -Lehrende, die von Google selbst, aus der Industrie oder von Universitäten veröffentlicht werden.

Eine Alternative bietet das OWASP WebGoat Project (http://www.owasp.org/index.php/Category:OWASP_WebGoat_Project). Hierbei handelt es sich um eine J2EE Anwendung, die eine sichere und legale Umgebung darstellt, um „Hacken“ in verschiedenen Lektionen auszuprobieren.

Fazit
In der Motivation des Themas wurde deutlich, dass die Gefahr von Angriffen auf Webanwendungen nach wie vor unterschätzt wird und es verschiedene Intentionen zum „Hacken“ gibt. Hierraus folgen verschiedene Tätertypen. Auch wenn es mit Sicherheit gutartige „Hacker“ gibt, sollte man im schlimmsten Fall davon ausgehen, dass der Täter kriminelle Absichten verfolgt und zudem hochmotiviert ist. Die OWASP Top 10 stellen zehn Ansatzpunkte für Webangriffe dar, wobei deutlich wird, dass einige Angriffe ähnlich und offensichtlich, aber Andere dafür unbekannt, einfach und trickreich sind. Für nahezu jede Angriffsart lässt sich ein Tool finden, dass durch Automatismus Arbeit erspart, allerdings niemals einen Menschen ersetzt. Da Menschen Lernen üblicherweise schwer fällt, bietet die Google Code University und das WebGoat Projekt Möglichkeiten, das Erlernen von Hacking zu vereinfachen.

4 Design und Architektur

Das folgende Kapitel gibt einen Überblick über das grundsätzliche Design und die Architektur des Frameworks.

Das Framework ist für die Durchführung von stream-mining Experimenten gedacht. Dabei werden prinzipiell unendliche Datenströme aus Quellen gelesen und kontinuierlich verarbeitet.

Diese Verarbeitung lässt sich unterteilen in eine Menge von Knoten, die untereinander verbunden sind und diesen kontinuierlichen Datenstrom bearbeiten. Ein Knoten ist dabei die kleinste funktionelle Einheit. Er erhält ein *event* aus dem derzeitigen Datenstrom, arbeitet auf diesem und verschickt es üblicherweise an weitere Knoten. Datenströme "fließen" also durch eine Reihe von Knoten (siehe Abbildung 6).

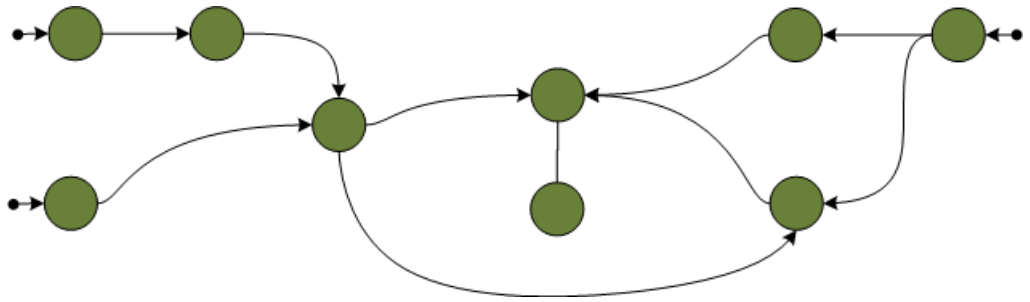


Abbildung 6: Aus drei Quellen (schwarze Punkte) geht ein Datenstrom durch eine Menge von Knoten (grüne Kreise). Knoten können dabei von mehreren Knoten einen Datenstrom empfangen, wie auch an mehrere Knoten weiterleiten

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 Experimentes. Die Knoten können verteilt in mehreren Knotencontainern liegen (siehe Abbildung 7).

Wie man anhand der Abbildung 7 sieht, ist der Datenfluss nicht nur auf einen Container begrenzt. Knoten in verschiedenen Knotencontainern können auch verbunden werden. Das Auffinden von Knoten übernimmt ein sogenannter Namensdienst. Dieser ist für eine Namensauflösung zuständig, wodurch der Knoten mit seiner Eingabe entfernt aufgerufen werden kann (siehe Abbildung 8).

Im Folgenden werden wir auf die einzelnen Komponenten und deren Schnittstellen eingehen. Zunächst wird der Aufbau und die Funktionsweise eines Knoten erläutert. Daraufhin wird näher auf den Datenfluss und die Kommunikation zwischen

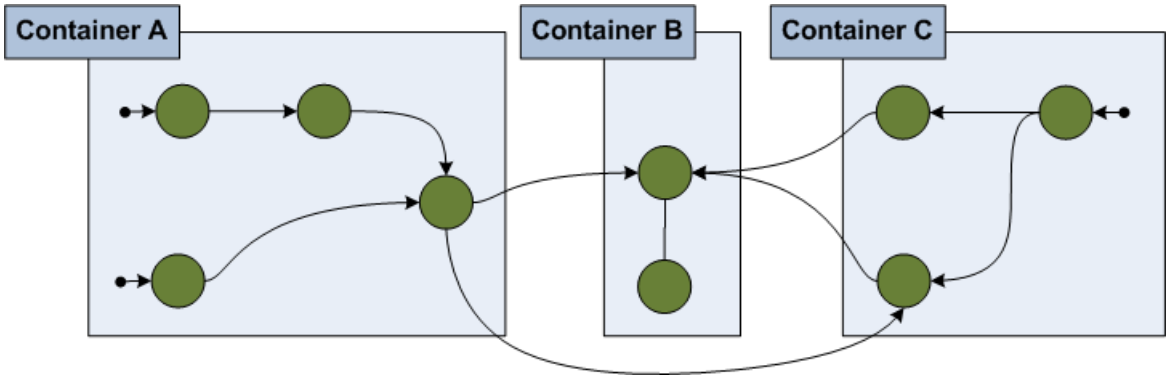


Abbildung 7: Knoten laufen auf verschiedenen Knotencontainern

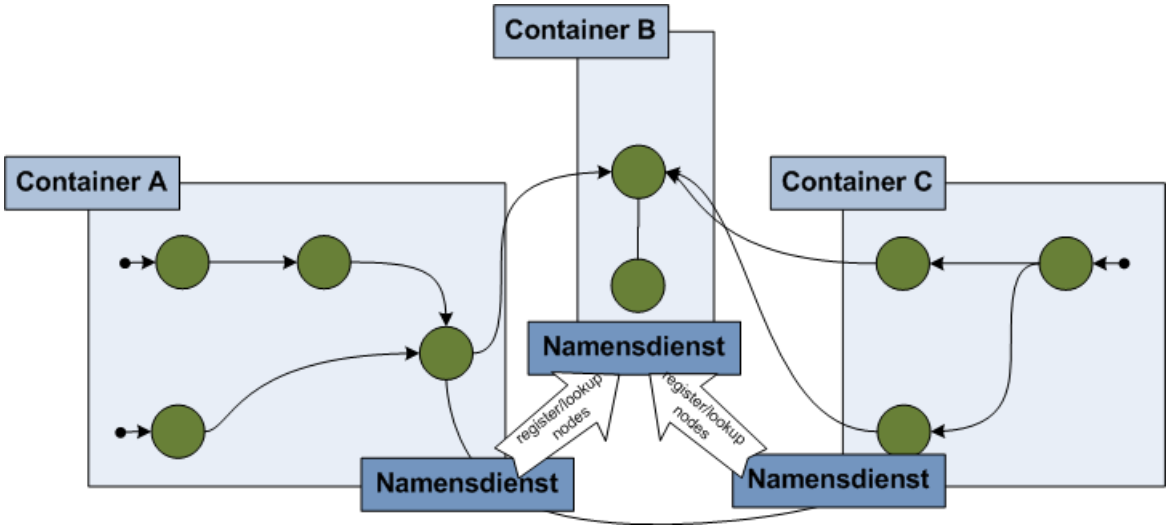


Abbildung 8: Die Knotencontainer fragen mit Hilfe eines übergeordneten Containers verteilte Knoten im Netzwerk an, damit seine lokalen Knoten mit den entfernten Knoten kommunizieren können.

den Knoten eingegangen. Nachdem die Funktionsweise und die Aufgaben des Knotencontainers erklärt wurden, wird zum Schluss noch beschrieben, wie der Namensdienst Knoten findet.

4.1 Knoten

Knoten sind die elementaren Bausteine aus dem im Framework ein stream-mining Experiment besteht. Ein Knoten bekommt eine Eingabe, verarbeiten diese und verschickt sie weiter als Ausgabe.

Die Entwicklung eines Knoten beschränkt sich dabei auf einige wenige und schlanke Schnittstellen:

edu.udo.cs.pg542.core.Node (Siehe 4.1.1) Die zu implementierenden Schnittstelle für jeden Knoten. Als abstrakte Standardimplementation wird *AbstractNode* (Siehe 4.1.3) im Packet *edu.udo.cs.pg542.core.impl* angeboten.

edu.udo.cs.pg542.core.NodeContext (Siehe 4.1.2) Mit dem *NodeContext* werden dem Knoten Informationen über seine Umgebung, sprich dem Knotencontainer, zur Verfügung gestellt. Dieser *NodeContext* wird dem Knoten bei der Initialisierung übergeben. Der *NodeContext* wird auch dazu benutzt ein *Event* weiterzureichen.

Alles, was man noch zusätzlich an Schnittstellen und Wissen braucht, um einen eigenen Knoten zu implementieren, sind dann noch die Datenflussschnittstelle *Event* (siehe 4.2.1) und die Klasse *Topic*, die die Möglichkeit liefert auf bestimmten Themenkanälen *Event*s zu verschicken und zu empfangen. (Für weitere Informationen siehe 4.2.4)

4.1.1 Node

Unsere Architektur sieht Knoten als funktionale Einheiten. Die Knoten funktioniert klassisch nach dem EVA-Prinzip (Eingabe, Verarbeitung und Ausgabe).

Die Logik der Verarbeitung findet dabei in der zu implementierenden Methode *execute* statt. Als Parameter wird dem Knoten seine Eingabe in Form eines *Event*-Objekts übergeben. Um den Aufruf dieser Methode kümmert sich das Framework. Nachdem die Eingabe verarbeitet wurde, hat der Entwickler drei Möglichkeiten:

- Das als Eingabe erhaltenen *Event*-Objekt (un)verändert weiterleiten.
- Ein neues *Event*-Objekt erstellen und weiterleiten.
- Kein Objekt weiterleiten. Das *Event* geht verloren.

```
1 public interface Node {
2
3   String getId();
4
5   void init( NodeContext nodeCtx );
6
7   void execute( Event event );
8
9 }
```

Listing 1: Schnittstelle zur Knotenimplementierung

Das Weiterleiten des *Event*s als Ausgabe passiert dabei über den *NodeContext* und seine *dispatch*-Methoden (siehe 4.1.2).

4.1.2 NodeContext

Eine der zu implementierenden Methoden der *Node* Schnittstelle ist die *init*-Methode, die als Parameter ein Objekt der Klasse *NodeContext* bekommt. Bei der Initialisierung eines Knotens wird diese Methode aufgerufen - also noch bevor die *execute*-Methode das erste Mal aufgerufen werden kann.

Über das *NodeContext* Objekt kann der Knoten mit seiner Umgebung, in die er eingebettet ist, interagieren. Folgende Methoden werden zur Verfügung gestellt:

dispatch(Event event)
: Das übergebende *Event* soll an die Nachfolgeknoten übergeben werden.

publish(Event event, Topic topic)
: Das übergebende *Event* soll auf den Kanal für das übergebene *Topic* gesendet werden. D.h., dass alle Knoten im Netzwerk, die sich für das gegebene *Topic* angemeldet haben, mit dem *Event* aufgerufen werden (siehe 4.2.4).

getCurrentTopics()
: Alle aktiven *Topic*s werden zurückgegeben. Ein aktives *Topic* ist ein *Topic* zu dem sich mindestens ein Knoten angemeldet hat (siehe 4.2.4).

subscribe(Topic topic)
: Der Knoten meldet sich zu dem angegebenen *Topic* an

unsubscribe(Topic topic)
: Der Knoten meldet sich zu dem angegebenen *Topic* ab

getParameters()
: Parametrisierte Knoten können hier ein *Parameters* Objekt holen, um die übergebenen Parameter zu bekommen (siehe 4.1.5).

getParameter(String key)
: Die Rückgabe eines einzigen Parameters mit dem Schlüssel “key”.

requiresParameter(ParameterRequirements requirements)
: Zur Validierung der Parameter kann diese Methode aufgerufen werden (siehe 4.1.5 und 4.1.4).

requires(String key, Class<?> clazz)
: Diese Methode sagt der Umgebung eines Knotens, welchen Datentyp er im Eingabe-*Event* erwartet (siehe 4.1.4).

delivers(String key, Class<?> clazz)
: Diese Methode sagt der Umgebung eines Knotens, welchen Datentyp er als Ausgabe liefert (siehe 4.1.4).

Für eine Vielzahl von Funktionalität wird also der *NodeContext* benötigt. Insofern wird der Knotenentwickler nicht darum herumkommen, die Referenz des *NodeContext* in der *init*-Methode zu sichern, um auf den *NodeContext* zur Laufzeit zugreifen zu können.

4.1.3 AbstractNode

Die Klasse *AbstractNode* wurde entwickelt, um den Umgang mit dem *NodeContext* komfortabler zu gestalten. *AbstractNode* implementiert die *init*-Methode, speichert die *NodeContext* Referenz und bietet direkte Methoden zu der Knotenumgebung an. So kann in der *execute*-Methode direkt *dispatch(event)* aufgerufen werden, statt einem *getNodeContext().dispatchEvent(event)*.

Ob nun das Interface *Node* implementiert wird oder von *AbstractNode* geerbt wird, ist für das Framework unerheblich. Insofern hängt das vom Geschmack des Entwicklers ab, wobei *AbstractNode* komfortable Methoden zum Preis einer festgelegten Vererbungshierarchie anbietet.

4.1.4 Validierung von Knoten

Frühzeitiges Erkennen von Fehlern in einem Experiment wird vom Framework aktiv unterstützt. Dabei gibt es zwei Fehlerquellen bei der Erstellung eines Experimentes, die zu einem Fehler führen können und u.U. erst spät und schwer zu finden sind.

Falsche Knoteneingabe
: Ein Knoten bekommt ein *Event*-Objekt, das nicht die erforderlichen Daten unter einem bestimmten Schlüssel enthält.

Falsche Parameter
: Einem Knoten wurden nicht die erforderlichen Parameter übergeben oder die Parameterwerte befinden sich außerhalb des erforderlichen Wertebereiches.

Solche Fehler können vermieden bzw. früh erkannt werden. Die Überprüfung führt das Framework durch. Dies erfordert aber auch, dass der Knoten dem Framework sagt, was er benötigt bzw. was er liefert. Dies passiert über den *NodeContext* in der *init*-Methode. Wenn die *init*-Methode mit dem *NodeContext* aufgerufen wird, muss der Knoten die passenden Methoden - noch in der *init*-Methode - aufrufen:

requires(ParameterConfiguration requirements)
: Dem *NodeContext* wird ein *ParameterConfiguration*-Objekt übergeben, mit dem der *NodeContainer* überprüfen kann, ob die vom Benutzer angegebenen Parameter ausreichend sind.

requires(String key, Class<?> clazz)
: Der Parameter von diesem Methodenaufruf verwendet den *NodeContainer*, um zu überprüfen, ob die Ein- und Ausgaben von verbundenen Knoten kompatibel sind. Ein Knoten, der beim Aufruf angibt, dass er unter dem Schlüssel "key" die Klasse "clazz" erwartet, wird nur Ziel von anderen Knotenausgaben, die die entsprechende *delivers*-Methode ebenfalls angegeben hat.

delivers(String key, Class<?> clazz)
: Mit dieser Methode gibt der Knoten an, dass er unter dem gegebenen Schlüssel die gegebene Klasse garantiert. Das Aufrufen dieser Methode ist zwingend, falls der Benutzer diesen Knoten mit einem Knoten verbinden möchte, der diese Eingaben erwartet.

Die Klasse *AbstractNode* erlaubt zusätzlich, dass diese Methoden direkt im Konstruktor aufgerufen werden können. Der Knoten kümmert sich dann automatisch um die Aufrufe bei der Initialisierung des Knotens.

4.1.5 Knotenparameter

Parameter sind ein wichtiger Bestandteil von Knoten, um sie zum einen wiederverwendbar zu gestalten, aber auch Lernknoten bestimmte Werte zu übergeben, die vom Kontext abhängen. Das Abfragen der Parameter erfolgt über die *getParameters()* bzw. *getParameter(String key)*-Methoden.

Wie schon in Kapitel 4.1.4 beschrieben, kann ein Knoten sowohl angeben, welche Parameter zwingend und welche optional sind, als auch einen Wertebereich und einen Typ angeben. Dazu benutzt man ein sogenanntes $ParameterConfiguration$-Objekt. Dies beinhaltet den Namen, Standardwerte und weitere Informationen. Da man es sowohl mit Zahlenwerten als auch Strings und anderen Datentypen zu tun haben kann, bieten wir für die Basistypen String, Integer, Float, Long und Double vorgefertigte Implementierungen an. Jede Implementierung beinhaltet speziell für den Typ geeignete Methoden. Die numerischen Typen besitzen beispielsweise die Möglichkeit obere und untere Grenzen für den Wertebereich festzulegen. Über die $requires$ Methode des $NodeContext$ kann eine $ParameterConfiguration$ hinzugefügt werden.

Wenn der dort definierte Parameter nicht als optional gekennzeichnet ist, muss dieser gesetzt werden; andernfalls wird bei der Validierung eine $ParameterException$ geworfen.

4.2 Datenfluss

Im Folgenden wird gezielt auf Aspekte der Kommunikation eingegangen: Zuerst auf den Datenaustausch zwischen Knoten, bzw. dessen Struktur, nachfolgend auf die verschiedenen Möglichkeiten Knoten zu verbinden.

4.2.1 Event

Um den Datenaustausch zwischen Knoten zu ermöglichen, wurde das *Event* eingeführt, welches als Container für Objekte fungiert. Diese werden unter einem Schlüssel innerhalb des Events gespeichert. Wichtig ist dabei, dass die gespeicherten Objekte serialisierbar sein müssen, da der Datenaustausch auch entfernt stattfinden kann.
Um ein Objekt in ein *Event* einzubetten benutzt man die *set*-Methode und übergibt dieser den Schlüssel, unter dem das Objekt später zu finden sein soll, sowie das zu speichernde Objekt.
Mit der *get*-Methode kann ein Objekt aus dem *Event* geholt werden. Als Parameter erhält diese Methode den Schlüssel, unter dem das Objekt zuvor gespeichert wurde. Der Rückgabetyp wird dabei vom Typ der Variable, die die Rückgabe speichern soll, abgeleitet; die Typisierung geschieht automatisch (siehe Listing 4.2.1).

```
1  String  x = event.get("x");   // Automatischer cast zu String
2  int     y = event.get("y");   // Automatischer cast zu int
3  int     z = event.get("x");   // ClassCastException, da 'x' vom Typ String ist
```

Listing 2: Ableitung eines Datentypes

Für die Übertragung einfacher Daten wie Strings, Integers, ... kann die *Event*-Datenstruktur unmittelbar genutzt werden. Es kann jedoch auch jederzeit ein (serialisierbares) Objekt eines eigens entworfenen Datentyps in ein *Event* gegeben und verschickt werden. Somit ist eine hinreichende Flexibilität für den Datenaustausch gegeben.

Ein weiterer Mechanismus dient zum Erfassen des sogenannten *Sendertrace*. Dieser enthält Informationen über den Weg, den ein *Event* durch das Netzwerk zurückgelegt hat. Diese Informationen können beispielsweise für das Debuggen bzw. Testen benutzt werden.
Der Sendertrace eines *Event*s setzt sich zusammen aus den Ids aller Knoten, die das *Event* verarbeitet haben. Zusätzlich gibt es noch eine Liste von Zeitstempeln, die angibt zu welchem Zeitpunkt ein *Event* an einem Knoten eingetroffen ist. Dabei gehört ein Zeitstempel mit dem Index i in dieser Liste zu dem Knoten mit dem Index i im

Sendertrace. Selbstverständlich lässt sich dieses Feature im Framework ausstellen, um eine Einschränkung der Performanz zu verhindern.

4.2.2 In Events verpackte Daten: Das Example

Im Folgenden werden die Datenstrukturen erläutert, mit denen Daten in ein *Event* gegeben und mit diesem an einen oder mehrere Knoten verschickt werden können.

Zur Übertragung von Attributnamen kann ein *FeatureSet* verwendet werden. Erweitert um eine Gewichtung pro Attribut ergibt sich das *WeightedFeatureSet*.
Sämtliche *Example* Typen - *Example*, *LabeledExample*, *WeightedExample* und *LabeledWeightedExample* - enthalten grundsätzlich ein *FeatureSet* und pro Attribut zugehörige Daten. *LabeledExample*, *WeightedExample* und *LabeledWeightedExample* enthalten darüber hinaus für das gesamte *Example* entweder genau ein Label, genau ein Gewicht oder beides. Grundsätzlich können diese *Example* Typen nicht von Hand erzeugt werden, sondern müssen über die *ExampleFactory* instanziiert werden.
Um eine Menge von *Example*s mit einheitlichem *FeatureSet* praktisch und ohne das ansonsten pro *Example* enthaltene *FeatureSet* zu verwalten, steht das *ExampleSet* zur Verfügung.

FeatureSet Das *FeatureSet* beschreibt eine Liste von Attributen. Es kann beispielsweise genutzt werden, um losgelöst von der Übermittlung von Daten einem Knoten mitteilen zu können, für welche Attribute ein *Example* Daten enthalten wird. Es kann sowohl mit einer fertigen *Collection*, als auch leer erzeugt und über die *set* Methode seriell mit Attributnamen versehen werden.

WeightedFeatureSet Um Daten unterschiedlicher Attribute nicht identisch zu behandeln, kann mit einem *WeightedFeatureSet* die Möglichkeit genutzt werden, einem Knoten eine Gewichtung für jedes Attribut mitzuteilen. Ein *WeightedFeatureSet* wird erzeugt, indem dem Konstruktor der Klasse entweder ein *default* Gewicht, oder ein *FeatureSet* zusammen mit einem default Gewicht übergeben wird. Spezielle Gewichte müssen dann einzeln mit der *set(String feature, Serializable weight)* Methode gesetzt werden. Mit dem default Gewicht wird dafür Sorge getragen, dass wenn nur einige wenige Attribute ein spezielles Gewicht haben nicht der Aufwand betrieben werden muss, das default Gewicht von Hand pro betroffenem Attribut zu setzen. Ferner ist das default Gewicht im *WeightedFeatureSet* nur genau einmal vorhanden, also nicht - wie bei explizit gesetzten Gewichten - pro Attribut und bietet so einen, wenn auch nur geringen, Speicherplatzvorteil.

Example Die Klasse *Example* repräsentiert eine Beobachtung (Beispiel), wie sie von Verfahren des maschinellen Lernens verwendet wird. Als solche bildet sie für Lernverfahren eine elementare Grundlage. Ein Beispiel besteht aus einzelnen Attributen (auch: Features). Jedes Attribut beschreibt eine Eigenschaft des Beispiels. So kann ein Attribut im Falle eines Wortvektors aussagen, ob ein Wort in einem Beispiel vorkommt oder nicht. In einem anderen Fall, kann es auch die Wetterlage beschreiben. Beispiele können als Funktion betrachtet werden, die jedem Attribut einen Wert zuordnet. Zwar erinnert dieses Verhalten an eine als *Map* bekannte Datenstruktur, jedoch wurden aus Performanzgründen für die Realisierung der *Example* Klasse Vektoren vorgezogen - ein Vektor für die Attributnamen und ein Vektor für die korrespondierenden Attributwerte.
Jedes *Example* ist in zwei Attributkategorien unterteilt: Reguläre- und spezielle Attribute. Die regulären Attribute sind oben beschrieben. Die speziellen Attribute können z.B. Metadaten eines Wortvektors beschreiben (wie id oder Zeitstempel), die nicht durch Vorkommen in den regulären Attributen überschrieben werden dürfen, oder sie beschreiben Attribute, auf denen nicht gelernt werden soll, wie z.B. Label_1 ... Label_n bei *multilabel* Lernverfahren (für den einfachen Fall *eines* ausgezeichneten Labels für ein gesamtes Beispiel sei auf die Klasse *LabeledExample* verwiesen). Attribute, ob regulär oder speziell, müssen eine eindeutige Namensgebung haben - es ist nicht möglich zwei Attribute mit identischem Namen und unterschiedlichen Werten im selben Beispiel zu verwalten. Bei jedem Setzen eines Wertes für ein Attribut wird der vorherige Wert jenes Attributs überschrieben.

Die Datenstruktur erlaubt typische Operationen auf Mengen wie Einfügen / Überschreiben eines Attribut-Wert-Paares (*set(Attributname, Attributwert)*, *setSpecial* für Spezialattribute), Abfrage eines Wertes (*get(Attributname)* bzw. *getSpecial*), oder Entfernen eines Attributes mitsamt korrespondierendem Wert (*remove(Attributname)* bzw. *removeSpecial*). Möglich ist auch das Abfragen von sämtlichen Attributen (Namen) über *getFeatures()* respektive *getSpecialFeatures()*.
Für weitere Informationen und Methoden sei an dieser Stelle auf die Dokumentation im Anhang hingewiesen.

WeightedExample, LabeledExample und LabeledWeightedExample Das Vorhandensein eines Labels und / oder eines Gewichts für ein Beispiel und die Handhabung dessen ist so elementar, dass diese Eigenschaft in eigenen Klassen abgebildet wird. Die Grundfunktionalität eines Beispiels bleibt vorhanden, wird jedoch erweitert. Hat ein Beispiel ein Label (z.B. „Spam“ oder „1“), so kann hierfür die Klasse *LabeledExample* genutzt werden, die zusätzlich zum *Example* Methoden zur Verwaltung eines Labels besitzt. Analog kann *WeightedExample* genutzt werden, wenn es erforderlich ist, dass ein Beispiel in seiner Gesamtheit ein Gewicht erhält. So kann ein Lerner, der Beispiele sukzessive erhält zwischen „wichtigen“ und „unwichtigen“ Beispielen unterscheiden. Soll ein Beispiel sowohl ein Label, als auch ein Gewicht erhalten, kann dafür die Klasse *LabeledWeightedExample* genutzt werden. Die Erweiterungen von *Ex-*

ample garantieren in gewisser Form das Vorhandensein eines Labels bzw. eines Gewichts, indem per *NoLabel-* und *NoWeightException* verhindert wird, dass diese Werte auf „null“ gesetzt werden. Beide Exceptions erben von *RuntimeException*, was einen try-catch-Block zwar ermöglicht, aber nicht notwendig macht.

ExampleFactory und Klassenstruktur Sämtliche oben beschriebenen Beispieltypen sind nicht als Klassen, sondern als Interfaces realisiert. Dies hat zum Einen den Vorteil, dass die konkreten Implementierungen leicht ausgetauscht werden können, zum Anderen ist dadurch in Java Mehrfachvererbung möglich - ein *LabeledWeightedExample* soll natürlich sowohl als *Labeled-* als auch als *WeightedExample* akzeptiert werden. Diese Forderung findet sich auch in der Klassenhierarchie wieder: *LabeledWeightedExample* erweitert *LabeledExample* und *WeightedExample*, die beide wiederum *Example* erweitern.

Die tatsächlichen Implementierungen der vier Example-Interfaces liegen in Klassen vor, die nach außen nicht sichtbar sind. Um die Instanziierungen zu kontrollieren und zu vereinfachen existiert die *ExampleFactory*. Diese Klasse besitzt statische create-Methoden für alle vier Beispieltypen (*create<Interfacename>*), die jeweils Objekte vom Typ der entsprechenden Interfaces erzeugen und zurückgeben. Bei der Instanziierung von Beispielen mit Label oder Gewicht, muss ein Wert ungleich „null“ direkt der create-Methode mitgegeben werden.

Code-Beispiele:

Erzeugen eines einfachen Beispiels:

```
Example myExample = ExampleFactory.createExample();
```

Erzeugen eines Beispiels mit Label und Gewicht:

```
LabeledWeightedExample myLWExample =
ExampleFactory.createLabeledWeightedExample(''Spam'', 0.2);
```

Ferner existiert für jeden der vier Beispieltypen jeweils eine weitere create-Methode, die zusätzlich ein Objekt des Typs *Example* akzeptiert. Mit diesen Methoden lassen sich

- Beispiele klonen:

  ```
  LabeledExample newLExample = ExampleFactory.
  createLabeledExample(oldLExample, oldLExample.getLabel());
  ```

- zu Beispielen ein Label, ein Gewicht oder beides hinzufügen (myExample ist vorher vom Typ *Example*):

```
WeightedExample myExample =
ExampleFactory.createWeightedExample(myExample, 0.8);
```

- von Beispielen ein Label, ein Gewicht oder beides entfernen (myExample ist vorher vom Typ *LabeledWeightedExample*):

```
Example myExample = ExampleFactory.createExample(myExample);
```

ExampleSet Das *ExampleSet* bietet die Möglichkeit, beliebig viele *Example*s in einem einzigen Objekt zu verwalten.

Für die Aufnahme eines *Example*s in ein *ExampleSet* ist die Methode *insertExample* vorgesehen, die als Parameter ein *Example*, oder eine Spezialisierung davon, erwartet. Hierbei werden die Werte und Spezialwerte des *Example*s extrahiert und in zwei Wertematrizen innerhalb des *ExampleSet*s überführt. Voraussetzung hierfür ist, dass die *FeatureSet*s des *Example*s und des *ExampleSet*s übereinstimmen. Bei Nichterfüllung wird dies mit einer *FeaturesNotEqualException* quittiert. Für die Benutzung der genannten Methode muss also von der das *ExampleSet* verwaltenden Instanz (in aller Regel ist dies ein Knoten) eventuell eine Angleichung der *FeatureSet*s durchgeführt werden, bevor mit *insertExample* eine Einfügeoperation durchgeführt werden kann. Dies geschieht über die *set(String feature, Serializable value)* des *Example*s bzw. die Methoden *addFeature(String feature, Serializable defaultValue)* und *addSpecialFeature(String specialFeature, Serializable defaultValue)* des *ExampleSets*. Im Falle des *Example*s wird ein Attribut unmittelbar mit einem zugehörigen Wert dem *Example* hinzugefügt. Da hingegen das *ExampleSet* bei der Angleichung von *FeatureSet*s keine solch harte, bindende Aussage über ein Attribut jedes *Example*s treffen sollte, wird lediglich ein default Wert vergeben.
Eine andere Möglichkeit, ein *Example* einem *ExampleSet* hinzuzufügen, ist die um einen Parameter erweiterte Methode *insertExample(Example example, Serializable defaultValue)*. Diese nimmt ein *Example* entgegen, mit dem grundsätzlich so verfahren wird, wie in der zuvor beschriebenen Methode. Der Unterschied liegt nun darin, dass bei einem noch nicht existenten Attribut mit dem Setzen des zugehörigen default Wertes reagiert wird, anstatt eine *FeaturesNotEqualException* zu werfen. Dies ist jedoch mit Bedacht zu verwenden - im Falle eines Wortvektors mag die Verwendung dieser zweiten Methode unbedenklich sein, da das Setzen eines default Wertes von „0“ für noch nicht vorhandene Wörter im Wortvektor sicherlich sinnvoll ist; bei semantisch divergenten Attributen innerhalb eines *FeatureSet*s jedoch wird in aller Regel die manuelle Erzeugung des *FeatureSet*s mit den zugehörigen default Werten vorzuziehen sein.
Die dritte und letzte Möglichkeit, ein *Example* einem *ExampleSet* hinzuzufügen, ist die Verwendung der Methoden *set(int example, int feature, Serializable value)* und *setSpecial(int example, int specialFeature, Serializable specialValue)*, welche eine Ansteuerung einzelner Zellen der Wertematrizen ermöglicht und auch innerhalb der zuvor beschriebenen *insertExample* Methoden verwendet wird. Die dazu nötigen Werte für

example und *feature* bzw. *specialFeature* lassen sich über die Methoden *size()*, welche die Anzahl der im *ExampleSet* enthaltenen *Example*s ausgibt und *getFeatures().size()* bzw. *getSpecialFeatures().size()*, welche die Anzahl der regulären und speziellen Attribute des *ExampleSet*s ausgibt, erfragen.

Zur Abfrage von *Example*s aus einem *ExampleSet* kann die Methode *extractExample(int example)* genutzt werden. Diese erzeugt intern über die *ExampleFactory* ein leeres *Example* und iteriert über die zugehörigen Werte der Wertematrizen. Falls hier ein Wert nicht gesetzt ist, wird auf den zugehörigen default Wert zurückgegriffen. Wenn darüber hinaus Label und / oder Gewicht für den übergebenen Index verfügbar sind, wird dieser / werden diese zusammen mit dem *Example* an die jeweilige Methode der *ExampleFactory* übergeben, so dass die *extractExample* Methode immer ein Objekt des speziellsten möglichen *Example* Typs zurückgibt.
Über einen zusätzlichen booleschen Parameter *alsoDefaultValues* kann gesteuert werden, ob bei der Erzeugung des *Example*s solche Attribute übersprungen werden sollen, für die in der Wertematrix kein Wert hinterlegt ist und stattdessen also eigentlich auf den default Wert zurückgegriffen werden müsste. Im Beispiel des Wortvektors würde auf diesem Wege ein *Example* für die Rückgabe erzeugt werden können, das nur die Wörter als Attribute enthält, die einen explizit gesetzten Wert haben - bei korrekter Verwendung würden also alle Attribute mit einem über den default Wert gesetzten Wert von „0“ nicht im erzeugten *Example* auftauchen.

4.2.3 Direkte Verbindung

Die erste Möglichkeit Knoten miteinander kommunizieren zu lassen, ist die direkte Verbindung. Dabei wird der Ausgang eines Knotens X unmittelbar mit dem Eingang eines Knotens Y verbunden. Die Anzahl der Direktverbindungen pro Knoten ist beliebig und nach oben lediglich durch die Verarbeitungsgeschwindigkeit des jeweiligen Knotens beschränkt.

Um ein *Event* über den Ausgang eines Knotens zu verschicken wird die *dispatch*-Methode des *NodeContext* (siehe 4.1.2) benutzt. Dabei wird das *Event*-Objekt dupliziert und an alle Nachfolgeknoten geschickt. So können sich Veränderungen in parallelen Knotensträngen nicht gegenseitig beeinflussen.

Direkte Verbindungen werden in einem Netzwerk von Knoten zum Zeitpunkt der Initialisierung der Knoten definiert und bei der ersten Benutzung aufgelöst. Sie können anschließend nicht mehr modifiziert werden - es liegt somit ein statisches Kommunikationsnetz vor. Dies stellt den Normalfall der Knotenkommunikation dar.

Wird hingegen eine dynamische Verbindung benötigt, bieten sich Themenkanäle an, die im Folgenden kurz erläutert werden.

4.2.4 Themenkanal

Die zweite Möglichkeit Kommunikation zwischen Knoten zu realisieren sind die sogenannten *Topic*s. Hierbei können Knoten sich zu gewissen Themen anmelden, um dann *Event*s zu empfangen, die wiederum von anderen Knoten über den zugehörigen Themenkanal gesendet werden.

Der wichtigste Unterschied zu der direkten Verbindung besteht darin, dass ein Knoten sich zur Laufzeit dynamisch bei einem Thema an- und abmelden kann, beispielsweise wenn ein Schwellwert überschritten wurde. Das An- und Abmelden geschieht über die *subscribe*- bzw. *unsubscribe*-Methode und das Veröffentlichen eines *Event*s über die *publish*-Methode des *NodeContext*s (siehe 4.1.2).

4.3 Container

Ein *NodeContainer* bildet die Laufzeitumgebung für eine Menge von Knoten. Hier werden die Knoten erstellt, initialisiert, gestartet und verwaltet. Die Aufgaben eines *NodeContainer*s lassen sich in zwei Phasen unterteilen: Die *startup*-Phase und die *dispatch*-Phase.

Die startup-Phase beschreibt den Ablauf eines soeben gestarteten *NodeContainer*s und unterteilt sich in sieben Schritte:

container-start
: Das Framework startet einen *NodeContainer*

container-input
: Die vom Benutzer übergebenen Daten - welche Knoten auf welchem Container laufen sollen - werden ausgelesen und die für den Container wichtigen Informationen an selbigen weitergegeben

node-instantiation
: Der *NodeContainer* instanziiert die ihm zugewiesenen Knoten

node-initialisation
: Für jeden Knoten wird ein eigenes *NodeContext*-Objekt erzeugt, in dem durch den Container alle angegebenen Parameter gesetzt werden

parameter-validation
: Es wird überprüft, ob die vom Benutzer übergebenen Parameter den Anforderungen der Knoten an die Eingaben entsprechen

link-validation
: Alle direkten Verbindungen werden validiert

node-start
: Für jeden Knoten wird ein Thread gestartet und die pro Knoten erforderlichen *Event*-Puffer werden erzeugt

kick-off
: Die vom Benutzer angegebenen Quell-/Startknoten werden mit einem leeren oder vom Benutzer vorgegebenen *Event* aufgerufen

Nach dem kick-off Schritt beginnt das Stream-Mining Experiment. Sollten bei der Validierung Fehler bemerkt worden sein, wird je nach Konfiguration entweder der Container gestoppt oder eine Warnung ausgegeben.

In der dispatch-Phase kümmert sich der Container um die Knotenausgabe, also um das Weiterleiten von *Event*s. Bei der ersten Verwendung einer direkten Verbindung wird versucht, die Knoten-Ids mit Hilfe des Namensdienstes (siehe unten) aufzulösen. War die Namensauflösung erfolgreich, wird in den *Event*-Puffer des Knotens das *Event* geschrieben und die Namensauflösung gesichert. Bei der Kommunikation über *Topic*s werden zunächst über den Namensdienst die an ein *Topic* angemeldeten Knoten ermittelt, um diese anschließend aufzulösen und deren *Event*-Puffer zu befüllen.

Der Namensdienst ist ein Register, in dem Knoten und ihre Erreichbarkeiten hinterlegt sind. Jeder Container hat einen lokalen Namensdienst, in dem lokale Knoten aufgeführt sind. Wenn ein zu erreichender Knoten nicht durch den lokalen Container aufgelöst werden kann, somit also in einem entfernten Container liegen muss, zu dem keine Verbindungsinformationen vorhanden sind, wird eine Anfrage an einen übergeordneter Namensdienst gestellt. Kann dieser die Verbindung ebenfalls nicht auflösen, wird ein weiterer übergeordneter Namensdienst hinzugezogen. Dies wird solange fortgesetzt, bis der höchstliegende Container erreicht wurde.
Die beschriebene Struktur nennen wir Kaskadierung von *NodeContainer*n, welche einen Containerbaum induzieren (siehe Abbildung 9).

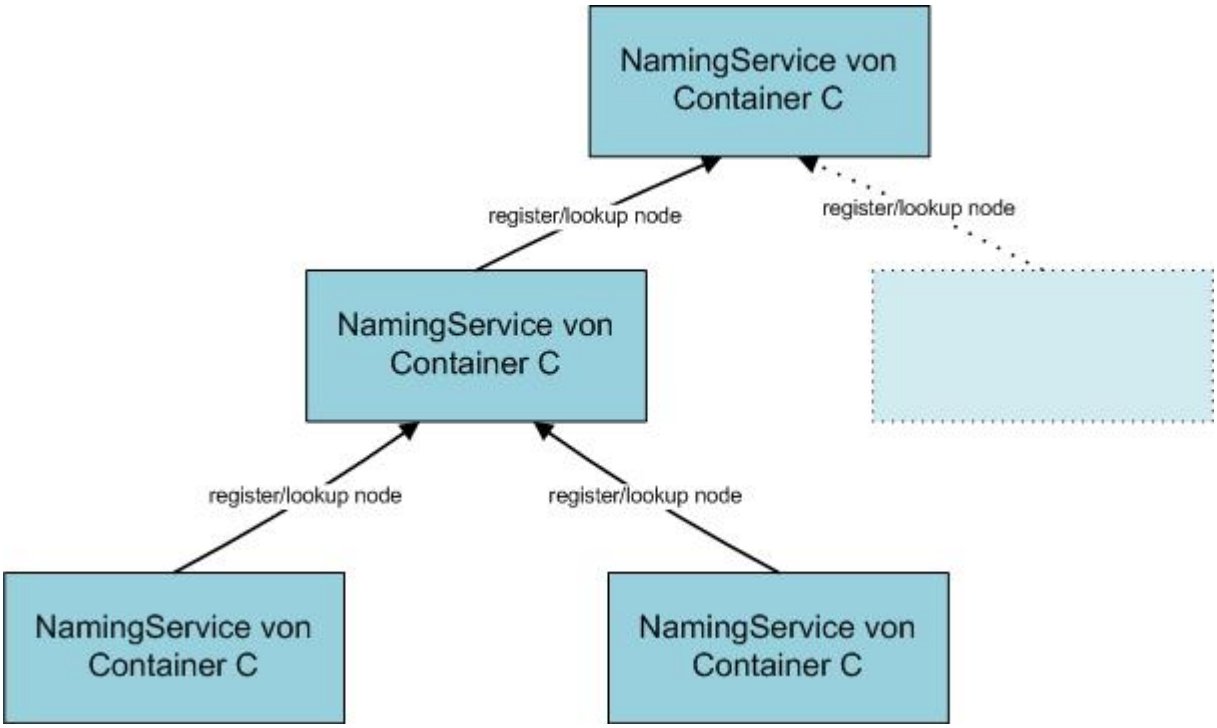


Abbildung 9: Namensdienste lassen sich mit anderen Namensdiensten zu einem Baum verschalten, so dass spätestens der Wurzelnamensdienst alle Referenzen auflösen kann

5 Use-Cases

Bei der Entwicklung eines generischen Frameworks ist es naheliegend, sich zunächst einige Anwendungsgebiete für das Endprodukt anzusehen. Dies hilft, benötigte Funktionalitäten und mögliche Problemstellen zu identifizieren.
Daher haben wir uns entschieden, zwei Anwendungsfälle genau zu beleuchten. Die Ergebnisse dieser Use-Case-Analysen sind im folgenden Kapitel zusammengefasst.

5.1 Intrusion Detection auf Basis von Logfiles

Im ersten Anwendungsfall sollen Angriffe auf Netzwerke anhand von *Logfiles* erkannt werden.

Problemstellung

Es sei ein Rechnernetz gegeben, in dem mehrere Server Dienste anbieten. Die Dienste haben gemein, dass sie jeweils in einem Logfile sämtliche Authorisierungsversuche protokollieren. Die Art des Logfiles ist jedoch stark von der jeweiligen Anwendung abhängig. Diese inhomogenen Datensätze dienen nun als Eingabe für geeignete Lernalgorithmen, durch die mögliche Angriffe aufgedeckt werden sollen.
Der Anwendungsfall ist in Abbildung 10 abstrahiert dargestellt.

Jeder angreifbare *Host* des zugrundeliegenden Netzwerks beherbergt einen lokalen Container mit Knoteninstanzen, die die auf dem Host gespeicherten Logfiles an das Knotennetz übermitteln. Desweiteren gibt es im Knotennetz einen globalen Container mit verarbeitenden Knoteninstanzen.

Um einen Angriff auf einen Server des Rechnernetzes anhand eines Logfiles zu erkennen, müssen die Attribute zueinander in Beziehung gesetzt werden. Mögliche Ansätze wären hierbei:

a. Zählen der Einträge pro IP innerhalb festgelegter Zeitschranken und Vergleich der Ergebnisse mit einem festgelegten Schwellwert. Zu aktive bzw. zu schnelle Benutzer könnten einen Hinweis auf Missbrauch, z.B. durch Bots, darstellen.

b. Zählen der verwendeten Benutzernamen pro IP und Vergleich mit einem festgelegten Schwellwert. Wechseln die eingegebenen Benutzernamen bezüglich einer IP-Adresse schnell, könnte es sich um einen Bruteforce-Angriff handeln.

c. Verhältnis der erfolgreichen und fehlgeschlagenen Login-Versuche pro Source-IP und Vergleich mit einem Schwellwert. Ein Missverhältnis dieser Einträge könnte auf einen Angriff hinweisen.

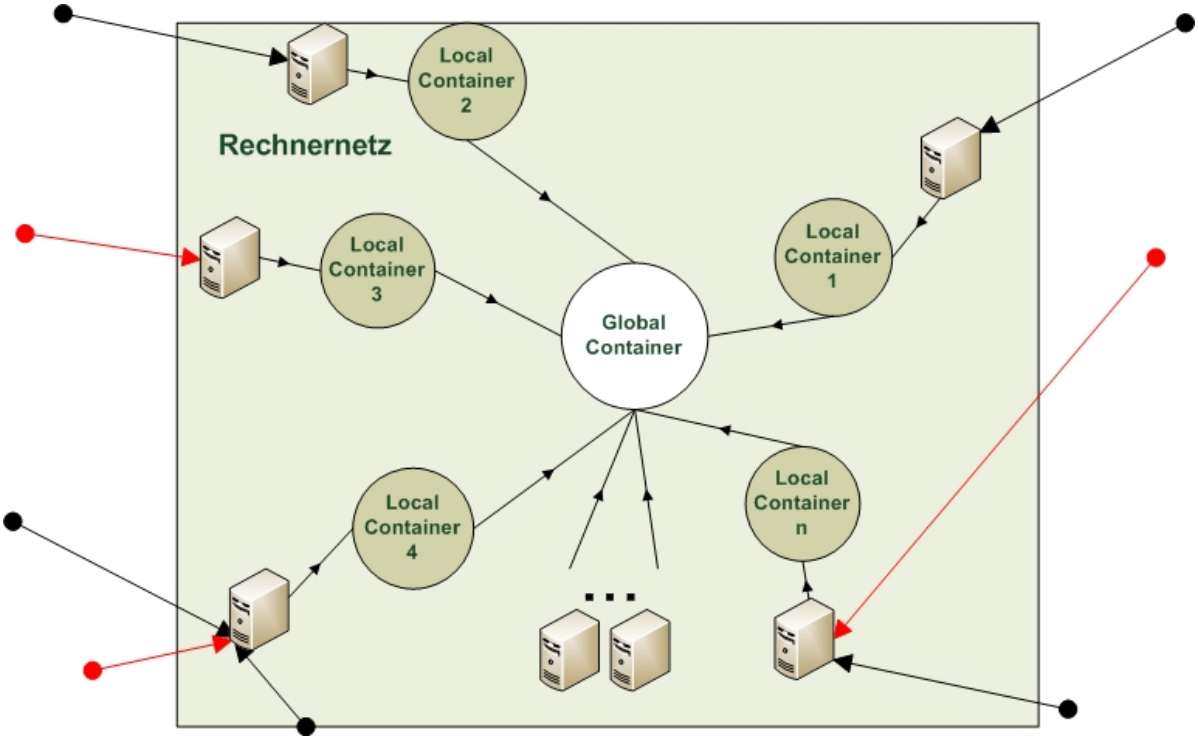


Abbildung 10: *Rechnernetz mit lokalen und einem globalen Container. Schwarze Pfeile stehen für erfolgreiche, rote Pfeile für fehlgeschlagene Logins*

d. Auslastung der Server untereinander (bei verteilten Anwendungen) vergleichen. Wird ein Server besonders häufig angefragt, könnte der Server gezielt attackiert werden.

Im vorliegenden Anwendungsfall wurde Ansatz c) ausgewählt, da er eine einfache, aber ausdrucksstarke Möglichkeit bietet, Angriffe zu erkennen. Des Weiteren könnte dieser Ansatz leicht mit Ansatz a) kombiniert werden, da die Einträge pro IP-Adresse für die Berechnung des Verhältnisses zwischen erfolgreichen und fehlgeschlagenen Logins ohnehin gezählt werden müssen.

Architektur

Für die Auswertung der Daten sind die folgenden Knoten innerhalb des Frameworks erforderlich:

- **Eingabeknoten**
  Das relevante Logfile wird durch einen *DataFetcher* überwacht, welcher den ersten Knoten und Lieferanten von Eingabedaten des konstruierten Netzwerks darstellt. Der DataFetcher überwacht registrierte Logfiles auf Änderungen und gibt diese zeilenweise an einen nachfolgenden Knoten zur Konvertierung weiter.

- **Parserknoten**
  Die Konvertierung der Rohdaten erfolgt durch einen Parser. Da die Inhalte stark von der jeweiligen Anwendung abhängig sind, muss der Parser speziell an das zu verarbeitende Logfile angepasst werden. Daraus ergibt sich der Vorteil, dass direkt eine Filterung der Rohdaten erfolgen kann und somit nur relevante Daten an nachfolgende Knoten weitergeleitet werden müssen. Das Produkt eines Parsers ist ein Beispiel, welches die relevanten Attribute der Rohdaten abbildet. Bezüglich des gewählten Ansatzes sind die Attribute *Timestamp*, *Source-IP*, *Server-IP*, *Service*, *Username* und *Success (Ergebnis des Logins)* von Interesse.

- **Lernknoten**
  Nachdem die Rohdaten in Beispiele umgewandelt wurden, dienen diese als Eingabe für Lernknoten. Je nach gewähltem Ansatz steht eine Vielzahl von Lernalgorithmen zur Auswahl. Falls mehr als ein Lernknoten vorgesehen ist, können auch mehrere Lernalgorithmen kombiniert werden.
  Für den gewählten Ansatz ist die Entscheidung getroffen worden, sogenannte *Hierarchical Heavy Hitters (HHH)* detektieren zu wollen.

- **Ausgabeknoten**
  Die gewonnenen Erkenntnisse sollen für die Erkennung bzw. Abwehr von Angreifern genutzt werden. Dies kann im einfachsten Fall durch die Benachrichtigung eines Administrators über vermutete Angriffe geschehen. Eine automatisierte Reaktion, wie das Verwerfen von Paketen, die von der IP-Adresse eines detektierten Angreifers stammen, ist denkbar, jedoch ist die Umsetzung reaktiver Komponenten vorerst nicht geplant.
  Um die produzierten Ergebnisse nachvollziehen zu können, wurde sowohl eine textuelle, als auch eine grafische Ausgabe implementiert.

Die Verteilung der oben genannten Knoten kann aufgrund der Containerarchitektur des Kommunikationsmodells prinzipiell beliebig geregelt werden. Eingeschränkt wird dies lediglich dadurch, dass ein Knoten genau einem Container zugeordnet sein muss. Eine feingranulare Aufteilung der Knoten auf eigene Container macht jedoch in der Regel keinen Sinn, da die Kommunikationslast - besonders im Fall von Containern auf unterschiefdlichen Stationen eines Computernetzwerks - so erhöht wird. Dies führt unter Umständen zu einer Verlangsamung des gesamten Ablaufs.

Während der Analyse des Anwendungsfalls wurden zwei Varianten näher betrachtet:

Variante 1 Bei der ersten Variante bestehen Container entweder aus Eingabeknoten und Parser oder aus Lern- und Ausgabeknoten. In diesem Szenario werden sämtliche aus den Rohdaten gewonnenen Beispiele an den sich im zentralen Container befindenden Lernknoten gesendet. Hier wird ein globales Modell der Dienste und Hosts erstellt.

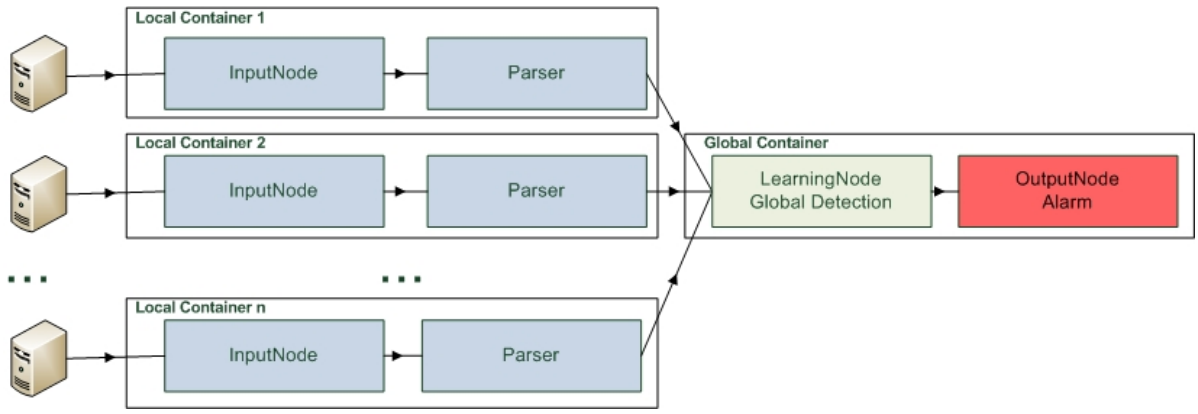


Abbildung 11: *Globale Verarbeitung der Beispiele.*

Variante 2 In der zweiten Variante soll die Netzlast verringert werden, indem nicht alle Beispiele an einen zentralen Container weitergeleitet werden. Dies wird erreicht, indem Lernknoten bereits in die lokalen Container integriert werden. Es werden also lokale Modelle erstellt, die bereits zur Angriffserkennung genutzt werden können. Im Gegensatz zur ersten Variante werden keine Beispiele an den zentralen Container gesendet. Ein globales Modell wird trotzdem erstellt, weil die lokalen Container ihre Modelle an den zentralen Container weiterleiten. Um dem grundsätzlichen Ziel - der Verringerung der Netzlast - zu genügen, werden die Modelle nicht kontinuierlich übertragen. Stattdessen könnte man Daten der lokalen Container in festen Zeitintervallen übermitteln, oder das globale Modell erst aktualisieren, wenn Änderungen in lokalen Modellen einen gewissen Schwellwert überschritten haben.

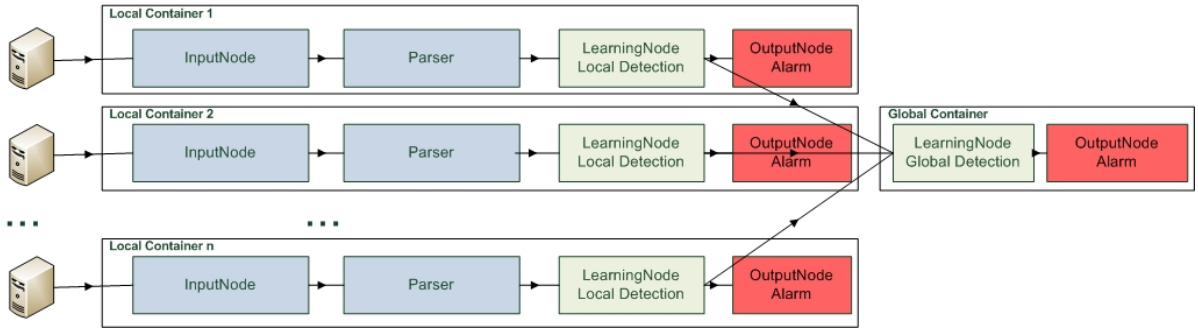


Abbildung 12: *Globale und lokale Verarbeitung der Beispiele.*

Umsetzung Es wurde die Entscheidung getroffen, die erste Variante des Architekturmodells abzubilden. Wenn Anomalien in einem Netzwerk mittels Hierarchical Heavy Hitters erkannt werden sollen, ist es notwendig, dass es einen globalen Container gibt, der Hierarchical Heavy Hitters im Kontext des Gesamtnetzwerkes berechnet. Dafür ist es wichtig, dass dem globalen Container keine Informationen entgehen, d.h. jeder lokale Container muss alle gesammelten Informationen an den globalen

Container weiterleiten, damit diese dort in einer globalen Datenstruktur analysiert werden können. Da zur Berechnung der Hierarchical Heavy Hitters alle Informationen gebraucht werden, macht es keinen Sinn, gewisse Informationen in den lokalen Containern herauszufiltern und nur bestimmte Ereignisse an den globalen Container zu reichen.

Datenstruktur

Bei der Entwicklung einer Datenstruktur zur Berechnung von Hierarchical Heavy Hitters innerhalb des Framework wurden zwei Ansätze entwickelt:

Variante 1 (TreeInTree) Es sollen Hierarchical Heavy Hitters auf Bäumen als Datenstruktur erkannt werden. Genauer gesagt soll ein Präfixbaum (*Prefix Tree* bzw. *Trie*) verwendet werden. In dieser Art von Bäumen werden die Blätter schrittweise bis zur Wurzel generalisiert. Die Generalisierung eines Elements *i* ist eine Obermenge die das Element *i* und weitere mit *i* verwandte Elemente enthält. Diese Obermenge ist wiederum Teilmenge einer weiteren Generalisierung. Die Wurzel des Präfixbaums ist die Menge *, die sämtliche Elemente beinhaltet.
Ein Beispiel für einen Präfixbaum ist in Abbildung 13 dargestellt: Hier ist ein Auszug aus dem lexikalischen Präfixbaum eines Mobiltelefons dargestellt. Es zeigt den besuchten Präfixbaum bei der Eingabe von *3-6-8-3*.

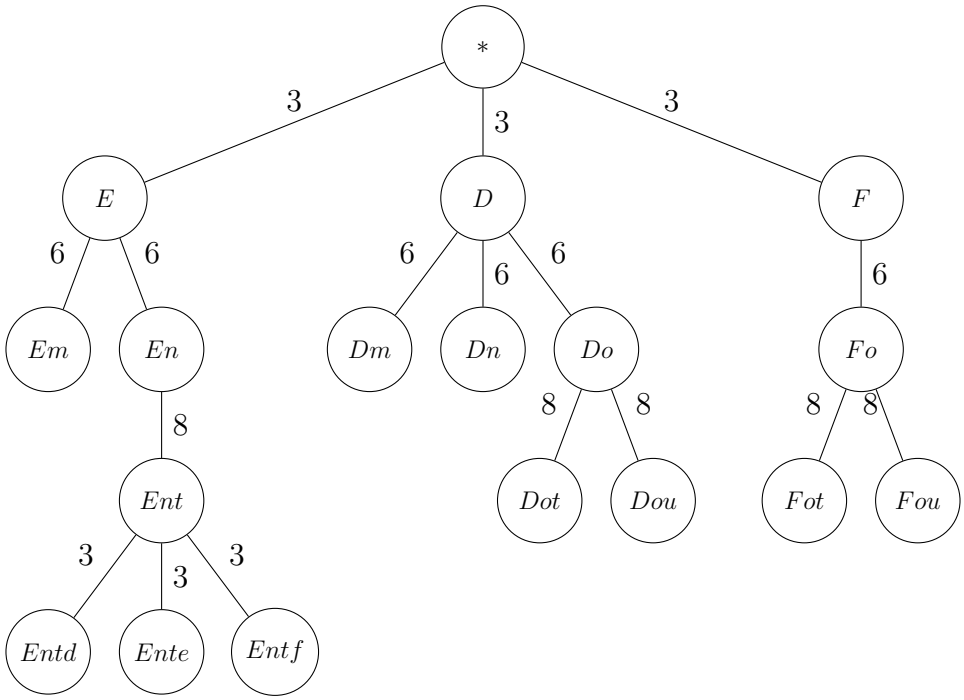


Abbildung 13: *Beispiel eines lexikalischen Präfixbaums*

In vorliegenden Anwendungsfall besteht das Problem, mehrdimensionale Daten abbilden zu müssen. Prinzipiell werden n-dimensionale Vektoren der Form $[d_1, ..., d_n]$ betrachtet. Die n Attribute des Vektors können als drei disjunkte Teilmengen angesehen werden:

- *Wertelemente* (*value-elements*): Diese Attributuntermenge dient als Eingabe für die Bewertung der Elemente durch den Lernalgorithmus.
- *ID-Elemente* (*ID-elements*): Diese Attributuntermenge wird als Label für die Knoten des Baums verwendet.
- *Metainformationen* (*meta-informations*): Alle Attribute, die nicht Wert- oder ID-Elemente sind, sind per Definition Metainformationen.

Durch diese Aufteilung wird die Dimensionen des Baums - bestimmt durch die Anzahl der als Label verwendeten Attribute - bereits verringert. Um keine Halbordnung über die m *ID-Elemente* definieren zu müssen, werden diese m *ID-Elemente* in eine feste Abarbeitungsreihenfolge gebracht.
Das folgende Vorgehen sieht vor, zunächst einen Präfixbaum für das erste *ID-Element* zu generieren. Der Baum hat genau t Blätter, wenn t die Anzahl der unterschiedlichen Attribute des ersten *ID-Elements* (d_1) in der Menge aller Beispiele ist. An jedes dieser Blätter wird ein weiterer Präfixbaum angehangen. Das Label dieser Bäume ist das zweite *ID-Element* (d_2). An einem Blatt d_1^i hängt ein Präfixbaum, der alle d_2^k enthält, die in Vektoren mit d_1^i vorkommen. An die Blätter der t Bäume mit dem zweiten *ID-Element* als Label werden dann wieder Präfixbäume mit dem dritten *ID-Element* als Label gehangen, und so fort.

Durch die Verkettung dieser Bäume bleibt ein attributübergreifender Zusammenhang erhalten, obwohl nicht mehr die n-dimensionalen Vektoren direkt betrachtet werden. *Metainformationen* können je nach Anwendungsfall an beliebiger Stelle gespeichert werden. Lediglich die *ID-Elemente* als Labels und die *Wertelemente* für die Bewertung der einzelnen Knoten müssen in jedem Knoten verfügbar sein.
Für den gewählten Realisierungsansatz mit den Attributen *Timestamp*, *Source-IP*, *Server-IP*, *Service*, *Username* und *Success* ergibt sich ein sechsdimensionaler Vektor. Als *ID-Elemente* werden *Source-IP* und *Server-IP* in dieser Reihenfolge gewählt. Das Attribut *Success* wird als *Wertelement* verwendet, was *Timestamp*, *Service* und *Username* zu *Metainformationen* macht. Denkbar sind auch Erweiterungen mit mehr *ID-Elementen* aus der Menge der *Metainformationen*. Sinnvoll erscheint besonders die Erweiterung, das Attribut *Service* als drittes *ID-Element* zu verwenden.

Variante 2 (Lattice) In der zweiten Variante wird als Datenstruktur ein Graph in Diamantenform, der eine Unterhalbordnung in einem Mengenverband beschreibt, verwendet. Der Graph stellt dabei, ähnlich wie die *TreeInTree*-Datenstruktur die unterschiedlichen Kombinationsmöglichkeiten der Generalisierung von Attributen dar, welche sich hierarchisch unterteilen lassen, wie z.B. IP-Adressen.

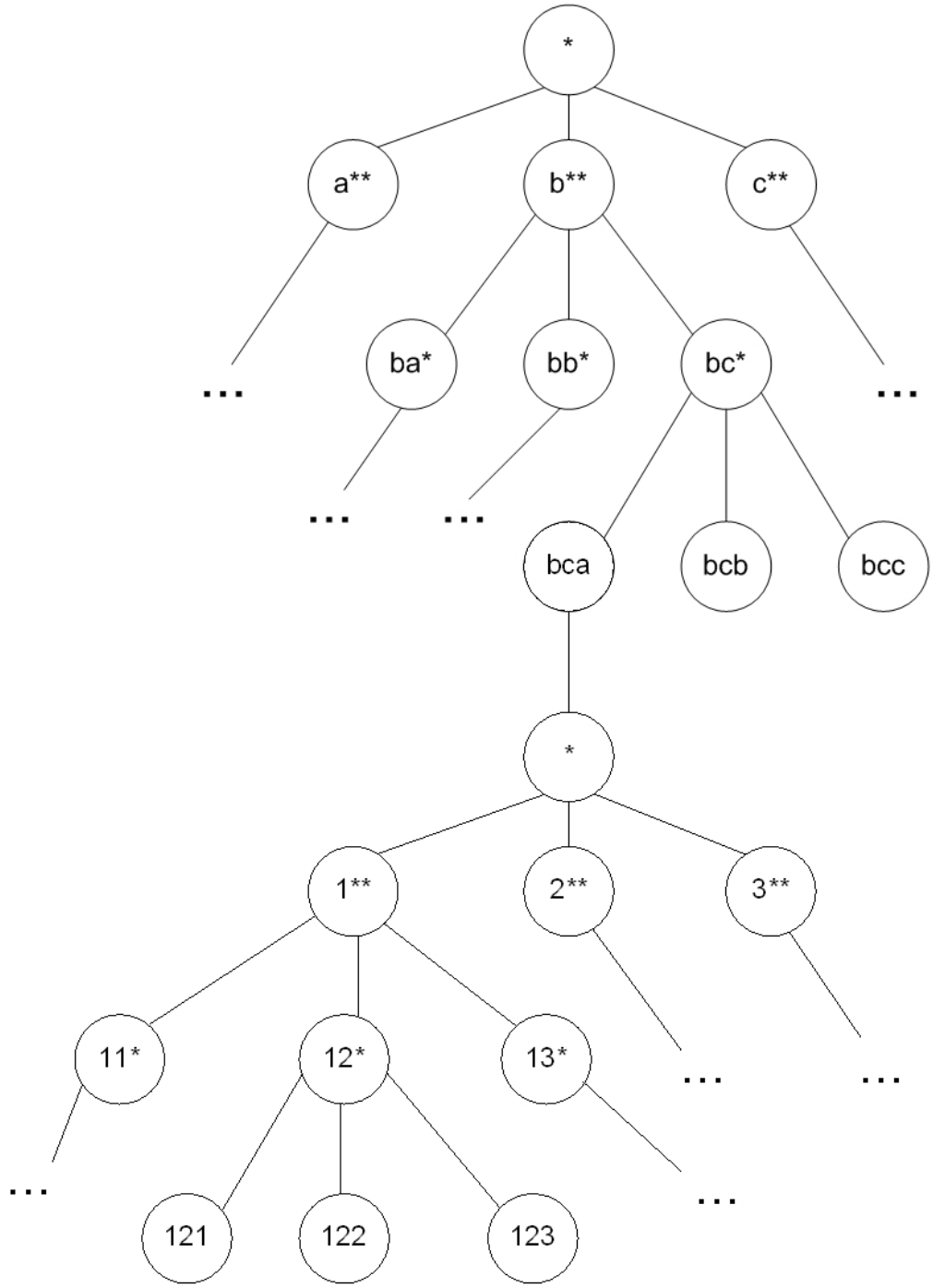


Abbildung 14: *Beispiel für einen zweistufigen Prefixbaum*

Ein wesentlicher Vorteil dieser Variante wäre, dass es bereits evaluierte Algorithmen zur Detektion von Hierarchical Heavy Hitters auf dieser Datenstruktur gibt. Im Paper von Cormode et al. [11] werden für dieses Problem der *Full Ancestry* und *Partial Ancestry* Algorithmus vorgestellt. Der Full Ancestry Algorithmus nimmt eine große

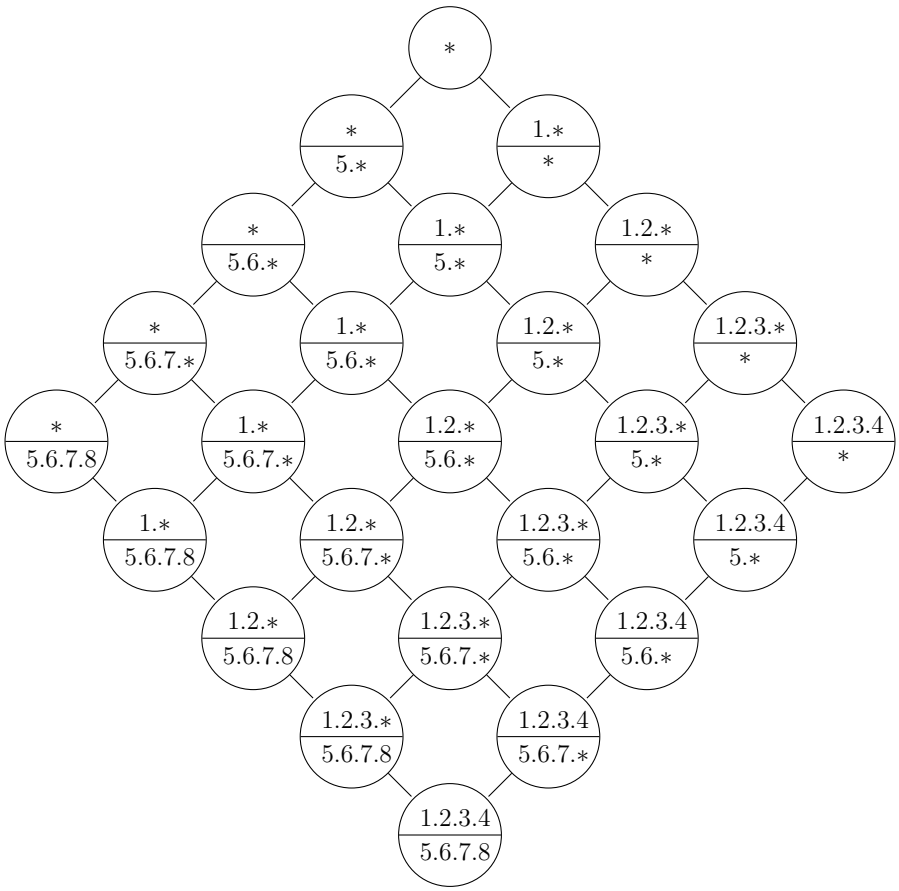


Abbildung 15: *Generalisierung einer IP-Adresse induziert eine Unterhalbordnung in Diamantenform mit zwei Attributen*

Datenstruktur in Kauf, d.h. er ist speicherintensiver, braucht dafür allerdings weniger Rechenzeit um die Elemente in die Datenstruktur einzufügen und Hierarchical Heavy Hitters auszugeben. Der Partial Ancestry Algorithmus hingegen braucht für das Einfügen neuer Knoten mehr Rechenzeit, da er versucht, so viele Knoten wie möglich in der Datenstruktur zu sparen und dabei trickreich die Informationen neuer Elemente auf bestehende Knoten verteilt, sofern dies möglich ist. Im behandelten Anwendungsfall haben Speicherbedarf und Rechenzeit die gleiche Priorität, da a) Hierarchical Heavy Hitters auf einem unendlich langen Stream detektiert werden sollen, aber nur endlich viel Speicher zur Verfügung steht und b) mit dem gleichen Argument neu ankommende Beispiele nicht lange gepuffert werden können, falls die Berechnung pro Element länger dauert als die Zeit zwischen der Ankunft zweier Elemente.

Umsetzung Die Entscheidung der für die Implementierung zu wählenden Variante fiel auf die zuerst genannte, da die Detektion der Hierarchical Heavy Hitters für eine erste Umsetzung so einfach wie möglich gehalten werden sollte. Zur Laufzeit

wird also, wie oben beschrieben, eine *TreeInTree*-Datenstruktur aufgestellt, die für *ID-Element* einen Unterbaum hält. Dabei werden die Knoten erst erzeugt, sobald ein neues Beispiel eintrifft, für das es noch keinen entsprechenden Knoten gibt. Ist bereits ein passender Knoten vorhanden, werden dessen Zähler, sowie die Zähler jedes Elternknotens, inkrementiert. Bei der HHH-Detektion werden die Knoten von der Wurzel ausgehend durchlaufen und es wird pro Knoten geprüft, ob sein Quotient aus erfolgreichen und gesamten Loginversuchen, vermindert um die Zähler seiner Kindknoten, die bereits Hierarchical Heavy Hitters sind, den Schwellwert überschreitet. Das heißt, dass für die Detektion eines HHH rekursiv geprüft wird, ob seine Kinder HHHs sind, bis die Nachfolgeknoten bekannt sind, die Hierarchical Heavy Hitters darstellen.
Die Erkennung von Hierarchical Heavy Hitters auf Basis des Verhältnisses zwischen fehlgeschlagenen Loginversuchen zur Anzahl der gesamten Login-Versuche, hat bei der Verwendung von Hierarchical Heavy Hitters allerdings auch neue Probleme verursacht. Bei einfacher Mittelung über die Quotienten der Kindknoten kann der Schwellwert beim Elternknoten nie erreicht werden. So können in einem Baum nur Blätter Hierarchical Heavy Hitter werden.

Dementsprechend wurde die Entscheidung getroffen, die Möglichkeit zur Verfügung stellen zu wollen, jeder Ebene der *Tree-in-Tree*-Datenstruktur einen eigenen Schwellwert zuweisen zu können. Da diese sehr hohe Anzahl an Schwellwerten niemals von einem Benutzer gesetzt werden könnte, soll sie gelernt werden. Dafür soll die Datenstruktur auf einem genügend großem Logfile, bei dem sicher ist, dass es keine oder kaum Anomalien und damit Angriffsversuche enthält, trainiert werden. Dabei sollen „übliche“ Schwellwerte auf jeder Ebene der Präfixbäume gelernt werden und Abweichungen von diesem Schwellwert Angriffe erkennbar machen.
Ein anderer denkbarer Ansatz - welcher momentan der Stand der Entwicklung ist - ist es, zusätzlich zum Schwellwert einen weiteren Wert zu übergeben, um den der Schwellwert pro Baumebene, von unten ausgehend, reduziert wird.

Implementierung

Die Java-Klassen der Implementierung sind auf mehrere Pakete verteilt. Dieser Abschnitt soll einen Überblick geben, an welcher Stelle welche Klasse zu finden ist und welche Funktionalität sie beherbergt. Details, wie z.B. die Funktionsweise der einzelnen Methoden der Klassen, können dann den Klassen selber entnommen werden, die über eine hinreichende Dokumentation verfügen.

- **Eingabeknoten** (Paket: *edu.udo.cs.pg542.node.input.file*)
  - *FileStreamReader.java*: Liest eine Datei (z.B. ein Logfile) Zeile für Zeile ein und leitet jede Zeile einzeln als String weiter.
  - *StdinStreamReader.java*: Verfahren analog, nur dass die Standardeingabe (stdin) und keine Datei beobachtet wird.
- **Parser** (Paket: *edu.udo.cs.pg542.node.input.transformation*)

  - *SSHParser.java*: Als Beispiel für einen Parser wurde ein SSH-Parser implementiert, der Zeilen eines SSH-Logfiles vom Eingabeknoten als String erwartet, diese mittels einem regulären Ausdrucks matcht und in Attribute einteilt und sie schließlich als Beispiele verpackt ausliefert.

- **Lernknoten**(Paket: *edu.udo.cs.pg542.node.learner*)
  - *HHHNode.java*: Erstellt eine Baumstruktur über die eingehenden Daten, auf der Hierarchical Heavy Hitters detektiert werden.

- **Ausgabeknoten**(Paket: *edu.udo.cs.pg542.node.output*)
  - *HHHPrefixTreeOutput.java*: Es wird eine grafische Ausgabe des durch den Lernknoten erzeugten Baums gezeichnet.
  - *SysoutOutput.java*: Für jedes in einem *Event* enthaltene Objekt wird die *toString()* Methode aufgerufen. Für die Präfixbaum Datenstruktur wurde diese Methode sinnvoll überschrieben.

- **Datenstruktur**(Paket: *edu.udo.cs.pg542.data.hhhprefixtree*)
  - *InnerTreeNode.java*: Beschreibt einen inneren Knoten des Präfixbaums eines Attributs und bietet Methoden zur Erkennung und Auflistung von Hierarchical Heavy Hitters auf sich und seinen Kindknoten.
  - *LeafTreeNode.java*: Stellt ein Blatt eines Präfixbaums dar und bietet somit die Möglichkeit einen weiteren Unterbaum zu halten, der einen Präfixbaum für ein anderes Attribut darstellt.

Während der Implementierung sind abstrakte Klassen entstanden, die gemeinsame Funktionalitäten ähnlicher Klassen abbilden: *AbstractWatcherThread.java* (Paket: *edu.udo.cs.pg542.nod* und *AbstractTreeNode.java* (Paket: *edu.udo.cs.pg542.data.hhhprefixtree*).

Tests

Die Klasse *SSHLogGenerator.java* (Paket: *edu.udo.cs.pg542.util*) erzeugt abhängig von vielfältig möglichen Parametrisierungen (wie z.B. einer Maske, die angibt, welche Bytes der IP randomisiert erzeugt werden sollen) eine Datei mit der Struktur eines SSH-Logfiles.

Die Klasse *LogfileTest.java* im Testpaket *edu.udo.cs.pg542.usecase.logfile* stellt ein komplettes auf den vorliegenden Anwendungsfall bezogenes Knotennetzwerk aus den implementierten Klassen zusammen. Ferner wurden erfolgreich Tests vollzogen, in denen Knoten in unterschiedlichen, physisch getrennt laufenden Containern instanziiert wurden (siehe *DistributedLogfileTest*Container.java* im selben Paket). In beiden Fällen kann die Ausgabe wahlweise textuell oder grafisch erfolgen.

5.2 Spamerkennung

Problemstellung Dieses Szenario behandelt den Fall der möglichst effizienten Spamerkennung für Emails auf verteilten Mailservern. Diese Mailserver bieten ihren Nutzern die Möglichkeit, erhaltene Emails als Spam zu markieren, sodass jeder Server über eine hinreichend große Menge von Trainingsdaten (klassifizierte Emails) verfügt. Die hinzugefügte Klassifizierung wird als *label* bezeichnet. Ziel des Verfahrens ist es, aus den Daten so zu lernen, dass neue Emails automatisch klassifiziert werden können.
Für die Klassifizierung von Trainingsmustern ist es unerlässlich, neben dem label eine Menge sogenannter Attribute (*features*) zu betrachten. Dies können im vorliegenden Fall spezielle Wörter oder Silben sein. Da es im Allgemeinen nicht effizient ist, mit gigantischen Mengen von Attributen zu arbeiten, können nicht alle Wörter des betrachteten Universums verwendet werden. Es stellt sich das Problem der Auswahl geeigneter Attribute (*feature selection*). Üblicherweise müssen hierzu alle Mailserver ihren gesamten Vorrat klassifizierter Emails auf einen globalen Server kopieren, auf dem dann die Güte möglicher Attribute berechnet und eine geeignete Menge von Attributen (*feature set*) zusammengestellt wird. Die Güte dieser Attributmenge muss dabei über einem vorgegeben Schwellwert liegen.
Das Ergebnis leitet der Server dann wieder zurück an die jeweiligen lokalen Mailserver, damit diese mit Hilfe der Attributmenge ihre Klassifikationslerner trainieren können. Das Problem, welches dabei notwendigerweise entsteht, ist die Versendung riesiger Datenmengen über das Netzwerk und die dadurch resultierende Auslastung. Um dem globalen Server die Änderungen des aktuellen Zustands mitzuteilen, müssen die lokalen Mailserver diese in regelmäßigen Zeitabständen (z.B. jede Woche/Monat etc.) übertragen. Nur durch dieses Vorgehen kann eine Anpassung des Spamfilters erreicht werden, da die Güte der ausgewählten Attribute von den zum Zeitpunkt der Berechnung vorliegenden Daten abhängt. Ändert sich die Zusammensetzung der Emailinhalte grundlegend, kann mit der vorhanden Attributmenge nicht mehr sinnvoll klassifiziert werden. Desweiteren ist es aus Sicht der Anwender möglicherweise unerwünscht, dass die Menge der Emails zunächst auf einen anderen Server kopiert und dort gesammelt werden müssen, um sie anschließend zu klassifizieren.

Der Geometrische Ansatz wählt hier einen Weg, der die Kommunikation zwischen den einzelnen Servern auf ein Minimum reduziert. Die Grundidee besteht darin, die Güte einer Attributmenge auf jedem Rechner lokal zu berechnen und nur dann die Verbindung zu anderen Servern zu suchen, wenn ein globaler Schwellwert überschritten wird. Das bringt aber ein Problem mit sich, denn der globale Schwellwert ist das Produkt einer beliebig komplexen Funktion (*Information-Gain-Funktion*, χ^2-Funktion, ...). Von den lokal berechneten Güten kann nicht auf die Größe des globalen Schwellwerts geschlossen werden.

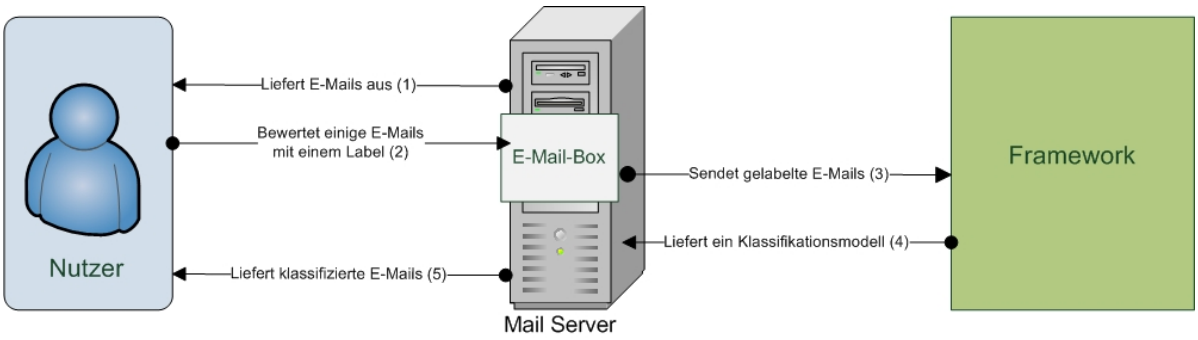


Abbildung 16: Spam Erkennung- Ablauf

Vorgehen Abbildung 16 erklärt die grundlegende Idee des Anwendungsfalls. Alle Nutzer angeschlossener Mailserver lesen ihre Mailboxen aus (1) und markieren einen Teil der darin enthaltenen Emails als Spam (2). Die Information wird zusammen mit der zugehörigen Email und einer Reihe von spamfreien Emails als Trainigsdatensatz einem Analyse-Framework übergeben (3). Das Framework verfügt über eine Reihe von Verarbeitungs- und Lernfunktionen mit deren Hilfe es ein Klassifikationsmodell erstellt und dieses an die Emailserver weiterreicht (4). Das erlernte Modell befähigt die Mailserver, neue und noch nicht markierte Emails als spamverdächtig zu klassifizieren (5). Kommt es zu Fehlklassifikationen, so kann der Nutzer irrtümlich als Spam bezeichnete Emails demarkieren, oder vice versa weitere Spammarkierungen vergeben. Diese Emails und ihr Spamstatus werden zum Zwecke der Fehlerkorrektur in der nächsten Iteration des Spamerkennungsprozesses als Teil der Trainigsdaten dem Analyse-Framework übergeben.

Ausgehend von Abbildung 17 wurden die erforderlichen Knoten des Spamerkennungssystems exemplarisch für einen Server (eine Instanz des Analysesystems) aufgeteilt. Am Anfang steht pro Instanz jeweils ein Emailserver, der kontinuierlich neue Emails empfängt. Hierbei markiert der Anwender die eintreffenen Emails und legt fest, ob es sich um eine Spammail oder um eine normale Email handelt. Dieses label wird im weiteren Verlauf mitgeführt.
Ein *observer*, muss den Eingang von Emails beobachten (indem z.B. einfach auf einem IMAP-Konto nach neuen Emails geschaut wird, oder es besteht bereits eine geeignete Schnittstelle zur Überwachung des Emailverkehrs).

Der angeschlossene *parser* generiert für jede neu eingetroffene Email eine zweispaltige Tabelle, bestehend aus den Wörtern, die in der Email vorkommen und einem zugehörigen Zahlwert, der die Häufigkeit des Wortes im Text angibt.
Diese sogenannten Wortvektoren dienen in Kombination mit den zugehörigen Spammarkierungen als Trainigsdaten für einen Klassifikationsalgorithmus. Hierfür wird eine geeignete Menge von Trainigsbeispielen in einem Cache gespeichert und zu einem temporären Traingsdatensatz aggregiert, welcher dem Lerner übergeben wird.

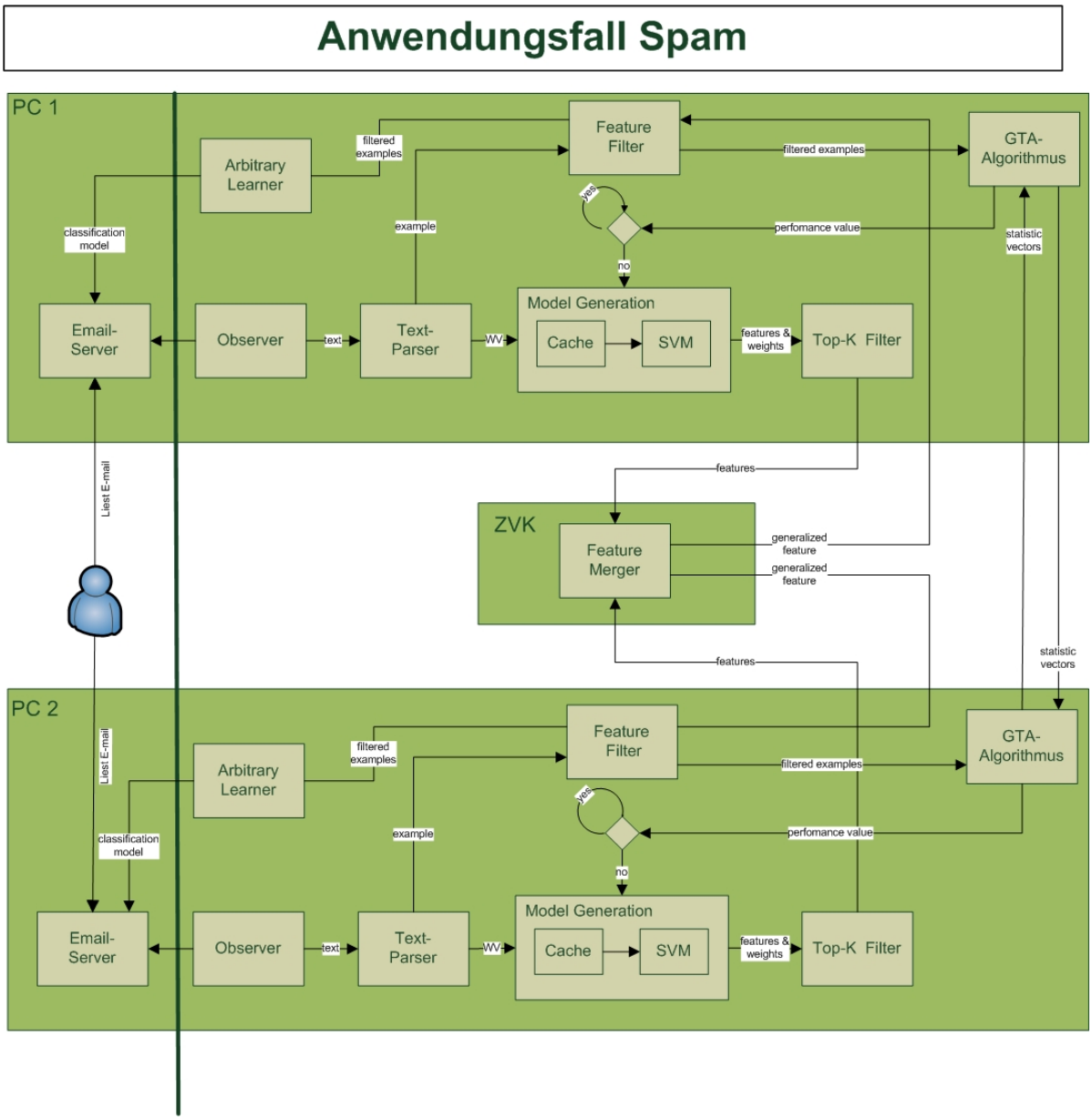


Abbildung 17: Spam Erkennung- Framework

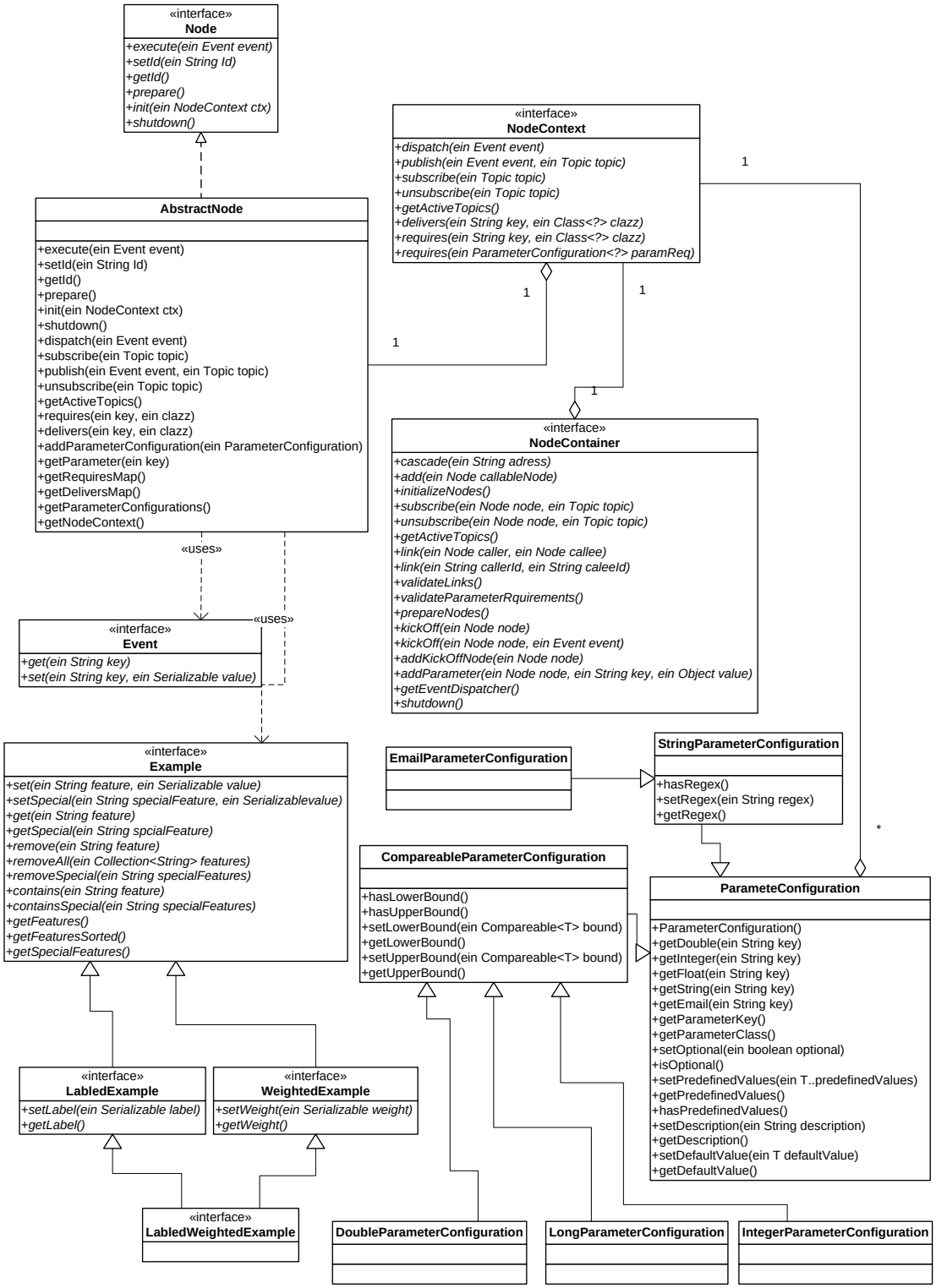
Der Lerner, beispielsweise eine *Support-Vector-Machine* (SVM) trainiert ein Modell und weist allen Attributen des Datensatzes Gewichte (*weights*) zu. Im Weiteren sind zunächst nur diese Gewichte, nicht aber das gelernte Modell von Interesse. Mit Hilfe dieser Gewichte können anschließend die k-wichtigsten Wörter aus den Trainigsdaten bestimmt werden. Jede Instanz nimmt diese Bestimmung vor und sendet ihre k-wichtigsten Wörter an eine zentralen Verteilerknoten (kurz ZVK). Dieser bildet aus den gesamten Wortmengen der Einzelinstanzen die Vereinigung und sendet diese Menge an alle Instanzen zurück. Eine Instanz besteht aus einem vollständigen Analysenetz, in Abbildung 17 für n=2 als PC1 und PC2 bezeichnet. Der aus der Vereinigung der k-wichtigsten Wörter von n Instanzen errechnete Wortvektor wird in jeder Instanz als Eingabe für einen Wortfilter (*featurefilter*) verwendet, um nur diejenigen Wörter für weitere Analysezwecke zu verwenden, die über das gesamte Analysesystem (d.h. alle Instanzen, PC1 - PCn) eine möglichst gute Indikatorfunktion erfüllen. Neu einlaufende Emails werden durch den so konfigurierten Wortfilter geführt und auf ihren Bestand an „nützlichen“ Worten reduziert. Solche Trainingsinstanzen werden in jedem Analysenetz einem beliebig wählbarem Lerner (*Arbitrary Learner*) zugeführt und ein lokales Klassifikationsmodell zur Erkennung von Spammails gelernt.
Die eigentliche Besonderheit des hier vorgestellten Systems liegt in der automatischen Erfassung der Dynamik eines auf unendlichen Datenströmen arbeitenden Spamerkennungssystems. Alle bisher erläuterten Schritte bis auf den finalen Klassifikationslerner dienen der Auswahl geeigneter Wörter zur Spamerkennung (*feature selection*). Ändert sich die Wortauswahl innerhalb der Emails grundlegend, müssen neue Indikatorwörter (*features*) gewählt werden. Diese Auswahl sollte aber nur dann von neuem ausgeführt werden, wenn dies auch notwendig ist, also die bisher ausgesuchten Wörter ihre Indikatorfunktion nicht mehr hinreichend erfüllen können. Einen entsprechenden Gütewert über einem verteilten System zu berechnen ist die Aufgabe des GTA-Algorithmus (kurz für *Geometric Threshold Approach*). Deshalb wird die Ausgabe des Wortfilters nicht nur an den finalen Lerner, sondern auch an die lokale GTA-Instanz weitergereicht.

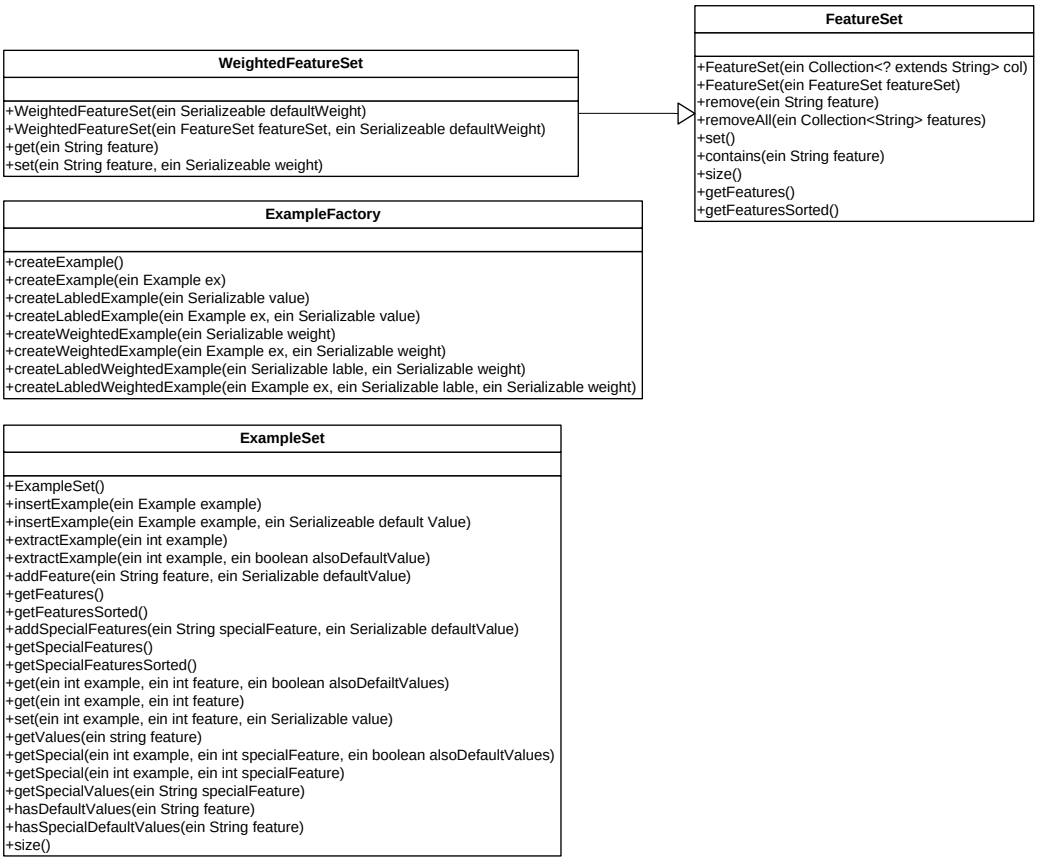
Der Wortvektor wird im nächsten Schritt zur Eingabe des GTA-Algorithmus. Dieser nutzt die Eingaben um über einem Fenster (*Sliding Window*) bestimmter Größe die Güte der Attribute aus lokaler Sicht zu berechnen. In bestimmten Abständen tauschen die einzelnen Instanzen einen aus den Attributvektoren eines Fensters errechneten Attributvektor untereinander aus und berechnen daraus einen neuen globalen Wert und eine globale Güte. Hierfür muss keine GTA-Instanz mehr als einen Attributvektor senden, der Kommunikationsaufwand bleibt klein. Das Ziel, die globale Güte unter einem bestimmten Schwellwert zu halten, erreichen die Instanzen des Algorithmus, indem sie zu jedem Zeitpunkt prüfen, ob sie bestimmte geometrische Eigenschaften bezüglich der lokalen Güte einhalten. Trifft dies für alle zu, bewegt sich die globale Güte in einem geeigneten Rahmen. Werden die lokalen Bedingungen an einer Stelle verletzt, werden alle anderen Instanzen benachrichtigt und eine Fehlerbehandlung wird durchgeführt.

Für den GTA-Algorithmus ist eine große Menge an unterschiedlichsten Erweiterun-

gen denkbar. Es könnten Varianten implementiert werden, die Attribute nicht nur auswählen, sondern gewichten oder automatisch auf neue Attribute prüfen, also den Datenstrom der Beispiele abhören und den Attributfilter anpassen. Für viele verteilte Datenquellen existiert auch eine Algorithmusversion, in der einer Instanz eine Koordinationsrolle zukommt, mit dem Effekt, dass bei einer notwendigen Fehlerbehandlung nicht mehr alle Instanzen benachrichtigt werden müssen.
Die Ausgabe des GTA-Algorithmus besteht aus einer Attributmenge, die von einem Klassifikationslerner verwendet werden kann, um Modelle zur Klassifikation unklassifizierter Emails zu erstellen.
Ausgehend von diesem Beispiel sei auf Kapitel 7.2.1 verwiesen. Dort wird das Verfahren formaler an einem Beispiel erläutert.

6 Klassenmodell




FeatureSet
+FeatureSet(ein Collection<? extends String> col)
+FeatureSet(ein FeatureSet featureSet)
+remove(ein String feature)
+removeAll(ein Collection<String> features)
+set()
+contains(ein String feature)
+size()
+getFeatures()
+getFeaturesSorted()
WeightedFeatureSet
+WeightedFeatureSet(ein Serializeable defaultWeight)
+WeightedFeatureSet(ein FeatureSet featureSet, ein Serializeable defaultWeight)
+get(ein String feature)
+set(ein String feature, ein Serializeable weight)
ExampleFactory
+createExample()
+createExample(ein Example ex)
+createLabledExample(ein Serializable value)
+createLabledExample(ein Example ex, ein Serializable value)
+createWeightedExample(ein Serializable weight)
+createWeightedExample(ein Example ex, ein Serializable weight)
+createLabledWeightedExample(ein Serializable lable, ein Serializable weight)
+createLabledWeightedExample(ein Example ex, ein Serializable lable, ein Serializable weight)
ExampleSet
+ExampleSet()
+insertExample(ein Example example)
+insertExample(ein Example example, ein Serializeable default Value)
+extractExample(ein int example)
+extractExample(ein int example, ein boolean alsoDefaultValue)
+addFeature(ein String feature, ein Serializable defaultValue)
+getFeatures()
+getFeaturesSorted()
+addSpecialFeatures(ein String specialFeature, ein Serializable defaultValue)
+getSpecialFeatures()
+getSpecialFeaturesSorted()
+get(ein int example, ein int feature, ein boolean alsoDefailtValues)
+get(ein int example, ein int feature)
+set(ein int example, ein int feature, ein Serializable value)
+getValues(ein string feature)
+getSpecial(ein int example, ein int specialFeature, ein boolean alsoDefaultValues)
+getSpecial(ein int example, ein int specialFeature)
+getSpecialValues(ein String specialFeature)
+hasDefaultValues(ein String feature)
+hasSpecialDefaultValues(ein String feature)
+size()

7 Lern-Algorithmen

7.1 Allgemeine Lernalgorithmen

7.1.1 SVM

Einführung *Support Vector Machines (SVMs)* werden als Klassifikatoren im Bereich der Mustererkennung verwendet. Generell werden Objekte in Mengen unterteilt und durch das *label* der jeweiligen Menge klassifiziert. Hierzu ist eine Trainingsphase nötig, in der bereits klassifizierte Objekte zum Aufbau der SVM genutzt werden. Die Objekte der Trainingsphase (Trainingsdaten) werden durch Vektoren dargestellt:

$$\{\vec{x_i}, y_i\} \qquad , i = 1, .., l \qquad , \vec{x_i} \in \mathbb{R}^d \qquad , y_i \in \{-1, 1\} \tag{1}$$

Zur Klassifizierung wird eine trennende Hyperebene genutzt, die zwischen den $\vec{x_i}$ mit label $y_i = -1$ und den $\vec{x_j}$ mit label $y_j = 1$ liegt.
Solche Hyperebenen können durch ihren Normalenvektor $\vec{w}$ und ihre Verschiebung vom Ursprung b beschrieben werden:

$$(\vec{w} \cdot \vec{x_i}) + b = 0 \tag{2}$$

Offensichtlich existieren mehrere trennende Hyperebenen, wie Abbildung 18 für den Fall linear trennbarer Daten zeigt.

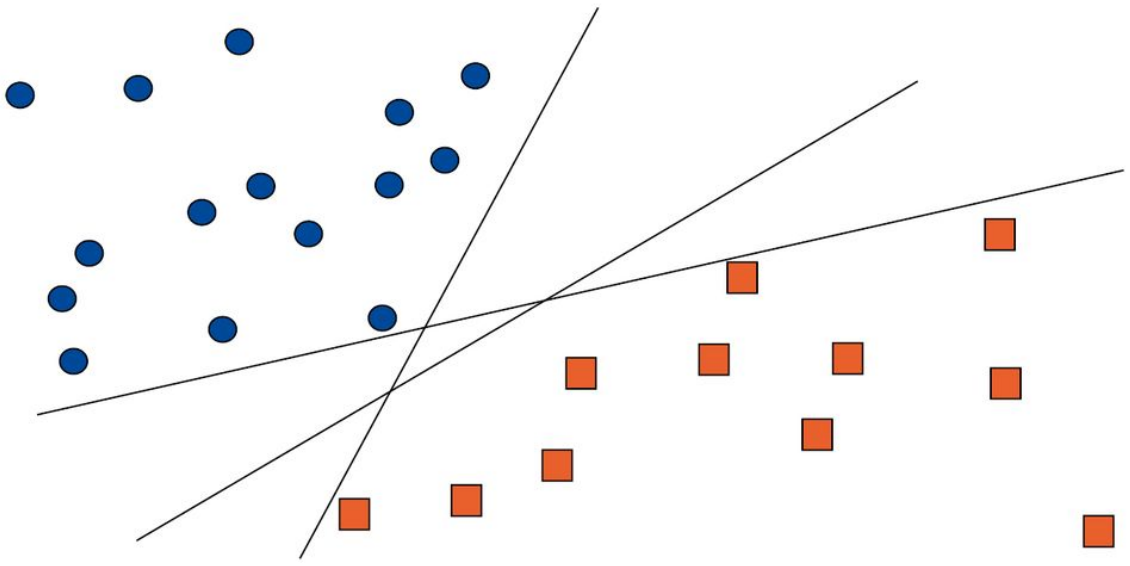


Abbildung 18: *mögliche linear trennende Hyperebenen*

Der Trainingsphase schließt sich eine Testphase an. Hier werden unklassifizierte Objekte einer Menge zugewiesen. Dafür wird geprüft, auf welcher „Seite“ der trennenden Hyperebene der Vektor $\vec{x}$ liegt, der das zu klassifizierende Objekt repräsentiert. Dies geschieht durch Bestimmung des Vorzeichens des Skalarprodukts von dem Normalenvektor der Hyperebene $\vec{w}$ und dem zu prüfenden Vektor $\vec{x}$. Die *Separationsregel* lautet dann:

$$f(x) = sign(\vec{w} \cdot \vec{x}) \tag{3}$$

linear trennbare Trainingsdaten Um eine möglichst gute Generalisierung zu erreichen wird für linear trennbare Daten die sogenannte „optimale Trennhyperebene“[22] verwendet. Eine Hyperebene trennt zwei Mengen optimal, wenn gilt:

- Trennung der Trainingsdaten geschieht ohne Fehler
- Abstand des Randvektors mit kleinster Entfernung zur Hyperebene ist maximal

Es ist leicht zu erkennen (siehe Abbildung 19), dass dies genau dann gilt, wenn der Abstand der Menge mit $y_i = -1$ gleich dem Abstand der Menge mit $y_j = 1$ ist.

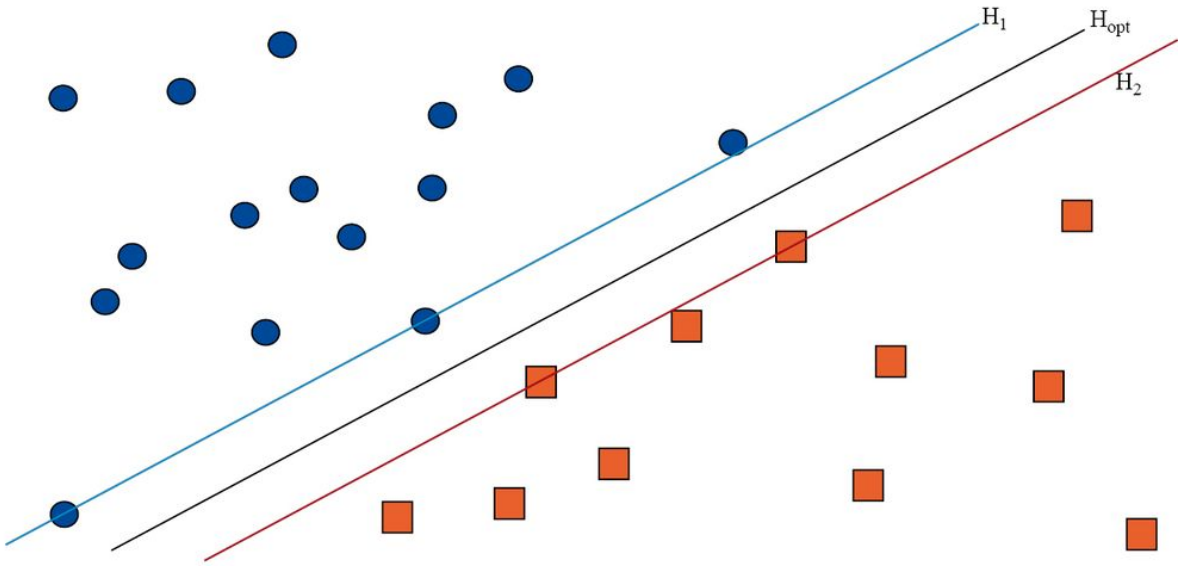


Abbildung 19: *optimale Trennhyperebene* H_{opt}

Vektoren mit geringster Entfernung zur Trennhyperebene - also diejenigen Vektoren die auf den Hyperebenen H_1 und H_2 liegen - werden Supportvektoren genannt und geben den SVMs ihren Namen. Für die Ebenen H_1 und H_2 kann eine Skalierung vorgenommen werden [18], so dass gilt:

$$\vec{w} \cdot \vec{x_i} + b = -1 \qquad , y_i = -1 \tag{4}$$
$$\vec{w} \cdot \vec{x_j} + b = 1 \qquad , y_j = 1 \tag{5}$$

Wenn wir den Abstand d_1 und d_2 berechnen, folgt daraus, dass der Rand $d_1 + d_2 = \frac{2}{\|\vec{w}\|}$ breit ist. Da per Definition dieser Rand maximal sein muss, ist die Länge des Normalenvektors $\|\vec{w}\|$ zu minimieren. Als Nebenbedingung ist zu beachten, dass keine Fehler vorkommen dürfen, also

$$\vec{w} \cdot \vec{x_i} + b \leq -1 \qquad , y_i = -1 \tag{6}$$
$$\vec{w} \cdot \vec{x_j} + b \geq 1 \qquad , y_j = 1 \tag{7}$$

bzw (7) zusammengefasst zu

$$y_i[\vec{w} \cdot \vec{x_i} + b] - 1 \geq 0 \qquad , \forall x_i \tag{8}$$

gilt.
Da dies ein Minimierungsproblem unter Berücksichtigung einer Nebenbedingung in Form einer Ungleichung ist, bietet es sich an, das Problem durch eine Lagrange-Funktion darzustellen [9]. Wenn die Lagrange-Multiplikatoren mit α_i bezeichnet werden, ergibt sich als neue Darstellung des Problems:

$$\mathcal{L}(\vec{w}, b, \alpha) = L_P = \frac{1}{2}\|\vec{w}\|^2 - \sum_{i=1}^{n} \alpha_i [y_i(\vec{w} \cdot \vec{x_i} + b) - 1] \tag{9}$$

Die Minimierung der Lagrange-Funktion geschieht unter Berücksichtigung der Nebenbedingung, dass die $\alpha_i > 0$ maximal sind. Wenn man nun den Gradienten bezüglich $\vec{w}$ und b berechnet, kann man die Lagrange-Funktion in das duale Problem umformen (für die einzelnen Schritte der Umformung kann [19] konsultiert werden):

$$L_D = \sum_{i=1}^{l} \alpha_i - \frac{1}{2} \sum_{i=1}^{l} \alpha_i \alpha_j y_i y_j \vec{x_i} \cdot \vec{x_j} \tag{10}$$

Das Minimieren der Lagrange-Funktion ist äquivalent mit dem Maximieren des dualen Problems unter der Berücksichtigung von $\alpha_i \geq 0$ und $\sum_{i=1}^{l} \alpha_i y_i = 0$.
Als letzte Vereinfachung des Problems brauchen nicht alle Vektoren der Trainingsdaten zur Bestimmung des Normalenvektors $\vec{w}$ betrachtet werden. Es reicht aus, $\vec{w}$ als Linearkombination aller $n < l$ Supportvektoren zu bestimmen. Dies wird klar, wenn

man gedanklich die nicht-Supportvektoren der Trainingsmenge entfernt oder verschiebt (natürlich nur außerhalb des Randes!). Solch eine Entfernung oder Verschiebung hat keinen Einfluss auf die Hyperebenen H_1 und H_2. Für eine mathematische Erklärung sei auf [23][S.129ff] verwiesen. Wenn die Anzahl der betrachteten Vektoren - d.h. der Anteil der Stützvektoren an der Gesamtmenge der Trainingsdaten - klein ist, verbessert sich außerdem die Generalisierungsfähigkeit der trennenden Hyperebene [23][S.135].
In der anschließenden Testphase kann nun (3) verwendet werden, die mit der Erkenntnis, dass $\vec{w}$ eine Linearkombination der Supportvektoren ist, zu

$$f(x) = sign(\sum_{i=1}^{n} y_i \alpha_i (\vec{x_i} \cdot \vec{x})) \tag{11}$$

umgeformt wird.

Nicht linear trennbare Trainingsdaten Für den Fall der nicht linear trennbaren Trainingsdaten kann eine optimale Trennhyperebene nicht wie im vorigen Kapitel beschrieben gefunden werden. Dies kann man leicht anhand der Definition der optimalen Trennhyperebene sehen, da eine solche Hyperebene keine Fehler bei der Trennung der Trainingsdaten machen darf.
Deshalb wird die optimale Trennhyperebene für den Fall von nicht linear trennbaren Hyperebenen anders definiert:

- maximaler Abstand zu den Stützvektoren
- geringste Fehlerquote auf den Trainingsdaten

Um die Fehler einzubeziehen werden Schlupfvariablen $\xi_i \geq 0$ eingeführt (siehe Abbildung 20), die den Rand verkleinern:

$$\vec{w} \cdot \vec{x_i} + b \leq -1 + \xi_i \qquad , y_i = -1 \tag{12}$$

$$\vec{w} \cdot \vec{x_j} + b \geq 1 - \xi_j \qquad , y_j = 1 \tag{13}$$

Wenn das duale Problem für diesen Fall aufgestellt wird [23][S.131ff], zeigt sich, dass dieselbe Funktion wie im linear trennbaren Fall maximiert werden muss, sich allerdings die Nebenbedingung ändert. Die α_i sind nun nicht nur nach unten durch 0 beschränkt, sondern auch nach oben durch eine frei wählbare Konstante C. Je höher C ist, desto stärker werden Fehler bestraft [9][S.14].
Die Testphase verläuft analog zum linear trennbaren Fall.

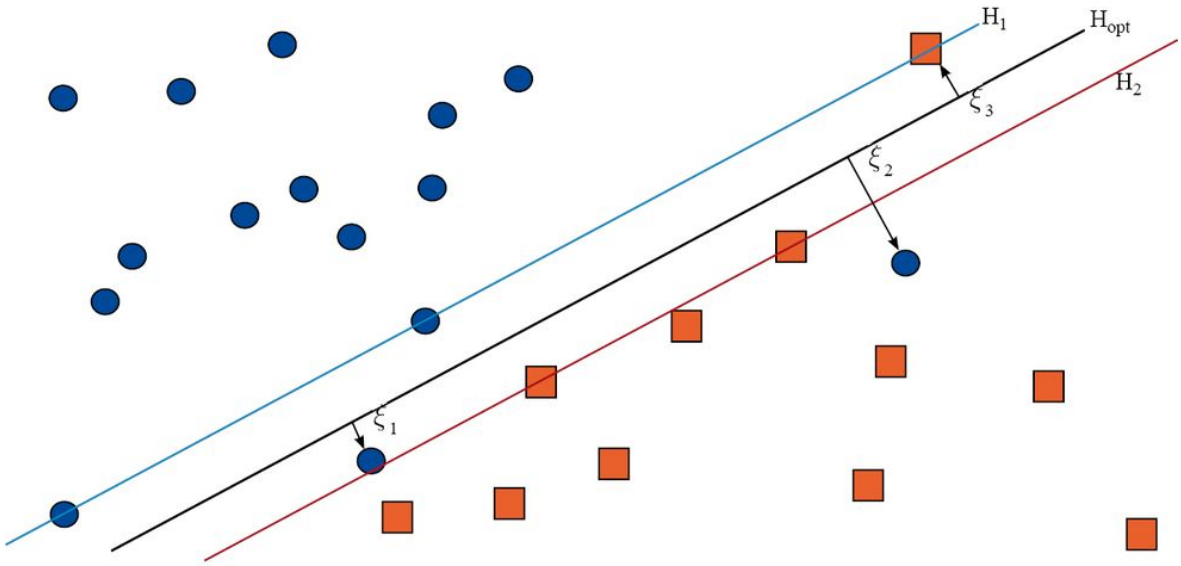


Abbildung 20: *nicht linear trennbare Trainingsdaten*

nicht lineare Hyperebenen In praxisnahen Beispielen werden linear trennbare Daten eher von geringer Bedeutung sein. Viel häufiger werden die Daten so stark streuen, dass die Verwendung von Schlupfvariablen nicht sinnvoll ist. Es existiert jedoch ein weiterer Ansatz um eine Hyperebene (bzw. in diesem Fall *Hyperflächen*) für nicht linear trennbare Trainingsdaten zu finden.
Anwendung findet hier eine Transformation der Trainingsdaten. Sind diese aus dem $\mathbb{R}^d$ und dort nicht linear trennbar, so können die Trainingsdaten in einen $\mathbb{R}^u$ mit $u > d$ überführt werden, in dem sie linear trennbar sind ($\vec{x_i} \in \mathbb{R}^d \longrightarrow \vec{z_i} \in \mathbb{R}^u$). Ein anschauliches Beispiel für den Fall $\mathbb{R}^2 \longrightarrow \mathbb{R}^3$ ist in Abbildung 21 dargestellt.
Da u auch unendlich sein kann, ist es meist nur theoretisch möglich, eine Hyperebene in solchen Räumen zu bestimmen [23][S.134]. Da es allerdings sowohl in der Trainingsphase bei der Lösung des dualen Problems (10) als auch in der Testphase in (11) ausreicht das Skalarprodukt zwischen den Vektoren zu bestimmen, können im Folgende *Kernel-Funktionen* verwendet werden. Eine Kernel-Funktion ist eine Faltung des Skalarprodukts zweier Vektoren.
Die Transformation in den Raum höherer Dimension ist so nicht notwendig:

$$K(\vec{x_i}, \vec{x_j}) = \vec{z_i} \cdot \vec{z_j} \tag{14}$$

Damit eine Funktion als Kernel-Funktion verwendet werden kann, muss sie der *Mercer-Bedingung* genügen [9].

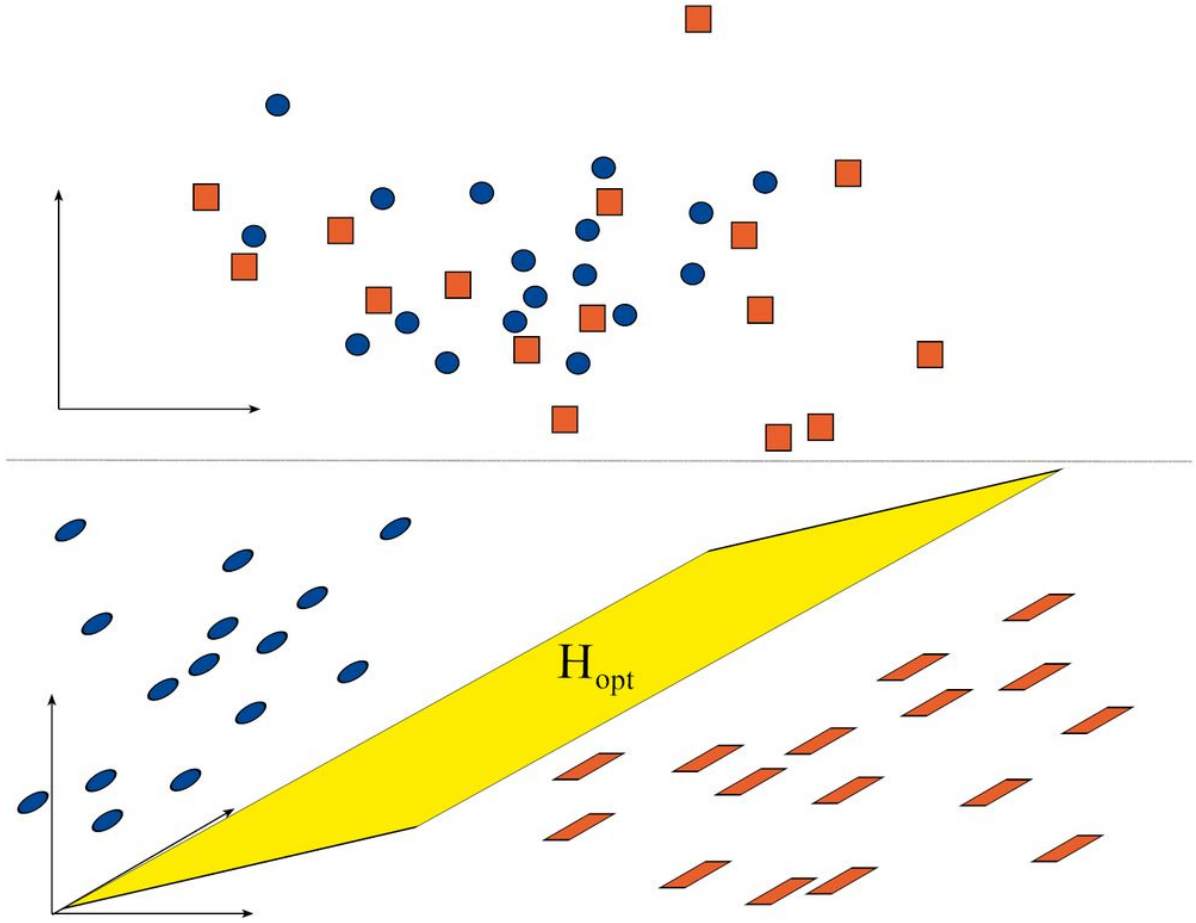


Abbildung 21: *Transformation von Trainingsdaten aus dem* $\mathbb{R}^2$ *in den* $\mathbb{R}^3$

7.1.2 Hierarchical Heavy Hitters

Motivation: 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 erstellen jedoch sekündlich hunderte von neuen, zu zählenden Transaktionen. Dies macht es schwierig "up-to-date" zu sein. Wer will seinen Kunden schon Videokassetten statt DVDs empfehlen? Aufgrund der ständig wachsenden

Datenmenge wird die Schwachstelle der klassischen Zählalgorithmen klar. Es werden Algorithmen benötigt, die live auf dem Strom der Transaktionen lernen.

Anwendung: Aber auch für die angestrebte Aufgabe der *Intrusion Detection* ist Zählen hilfreich. Bei der Analyse von IP-Paketen kann durch Zählen bestimmter Pakete und deren Eigenschaften ein möglicher Angriff erkannt werden.
Ein beliebter Angriff auf ein Netz ist die sogenannte *Denial of Service* Attacke. Ziel dieser Attacke ist es 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. Danach wartet der Server 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 *timeout* aufgibt. Da die Resourcen 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 statt 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 über Ports und/oder Empfängeradresse hinzunimmt. Für diese hierarchische Gliederung werden leicht angepasste Algorithmen benötigt.

Ein weiteres Anwendungsgebiet wird im ersten der betrachteten Anwendungsfälle näher beschrieben (siehe Kapitel 5.1).

Lossy Counting Algorithmus **- einfaches Zählen:** 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 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 dieses überhaupt ausgegeben wird und zum anderen ein maximal erlaubter 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 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

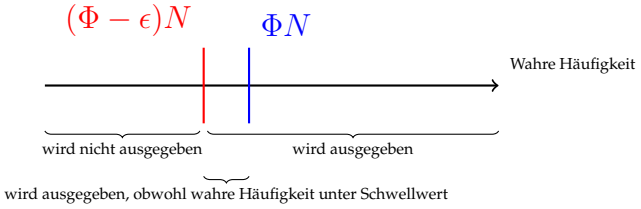


Abbildung 22: Lossy Counting Garantien

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.

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.

Um dies zu gewährleisten geht der Lossy Counting Algorithmus folgendermaßen vor:

Der Datenstrom wird in gleich breite Fenster der Breite $w = \lceil \frac{1}{\epsilon} \rceil$ aufgeteilt. Das bedeutet insbesondere, dass das derzeitige Fenster $b_{current} = \lceil \frac{N}{w} \rceil$ ist.

LossyCounting.insert(e, n) Ein eintreffendes Element e aus dem Datenstrom wird nun mit der *insert*-Methode in die interne Datenstruktur $\mathcal{D}$ eingefügt. In dieser Datenstruktur gibt es Einträge in der Form (e, f, Δ). Wird e neu eingefügt, dann wird f auf 1 und Δ auf das derzeitige Fenster $b_{current}$ gesetzt. Ist e schon in $\mathcal{D}$ enthalten, dann wird f einfach nur inkrementiert.

```
1 Insert( Element e, int count ) {
2   if( D.contains(e) )
3     D.get(e).f += count
4   else
5     D.put( new Entry(e,1,b_current-1) )
6 }
```

Listing 3: Insert-Methode des Lossy Counting Algorithmus

Δ wird also nach dem ersten Einfügen nicht mehr verändert. Trotzdem ist dieser Wert wichtig, um unsere Fehlergarantien sicherzustellen. Während f unsere geschätzte Häufigkeit von e darstellt, ist Δ unser maximal möglicher Fehler in f. Das wird aus den nächsten beiden Methoden des Lossy Counting Algorithmus ersichtlich.

LossyCounting.compress() Würden wir nur die *insert*-Methode verwenden, wäre Lossy Counting linear im Speicher, f wäre nicht geschätzt, sondern die tatsächliche Häufigkeit und Δ als maximaler Fehler wirklich sehr pessimistisch. Die *compress*-Methode stutzt die Datenstruktur $\mathcal{D}$, um Speicher zu sparen. Dies passiert am Ende eines jeden Fensters. Alle Elemente $e \in \mathcal{D}$ werden dabei überprüft, ob ihre geschätzte Häufigkeit f addiert zu ihrem maximalen Fehler Δ größer ist, als die derzeitige Fensternummer $b_{current}$.

```
7  Compress() {
8    for( Entry entry : D ) {
9      if( entry.f + entry.Δ ≤ b_current )
10       D.remove(entry)
11   }
12 }
```

Listing 4: Compress-Methode des Lossy Counting Algorithmus

LossyCounting.output(Φ) Der Benutzer kann nun zur Laufzeit einen Schwellwert Φ angeben und erhält als Rückgabe alle Elemente im Datenstrom mit den oben genannten Garantien.

```
13 Output( double Φ ) {
14   for( Entry entry : D ) {
15     if( entry.f ≥ (Φ - ε)N )
16       result.add( entry )
17   }
18   return result
19 }
```

Listing 5: Output-Methode

Sicherstellung der Lossy Counting Garantien Wieso werden mit diesen drei Methoden die Fehlergarantien sichergestellt? Das liegt an der Bedingung, wann Elemente aus der Datenstruktur $\mathcal{D}$ gelöscht werden:

$$f + \Delta \leq b_{current}$$

Δ wird nach dem Einfügen eines Elementes auf $b_{current}$ gesetzt und dann nicht mehr verändert. Da $b_{current}$ nach jedem Fenster inkrementiert wird, muss f im Durchschnitt auch in jedem Fenster einmal vorkommen. Ansonsten wird das Element gelöscht. Das bedeutet vor allem, dass ein Element welches nicht in $\mathcal{D}$ enthalten ist zwar schon

einmal enthalten gewesen sein kann, aber es ist sichergestellt, dass es nicht häufiger als die Anzahl der bisherigen Fenster im Datenstrom enthalten war. Da $b_{current} = \lceil \frac{N}{w} \rceil$ und $w = \lceil \frac{1}{\epsilon} \rceil$ ist, ist unsere geschätzte Häufigkeit f höchstens ϵN zu klein, dies führt zu unseren Garantien.

Hierarchical Heavy Hitters 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 bei der Anwendung erwähnt, kann es vorteilhaft sein, nicht nur über ein Element selbst, sondern auch über seine hierarchischen Teilstrukturen Aussagen zu treffen. Wir wollen also auch Angriffe aus Teilnetzen zählen und nicht nur einzelne IP-Adressen.

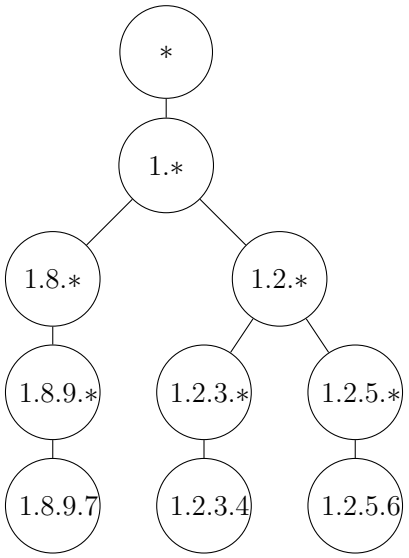


Abbildung 23: 3 IP-Adressen aus einem Datenstrom mit ihren verschiedenen Hierarchiestufen

Das naive Zählen in allen Hierarchiestufen würde uns dabei aber nicht weiterhelfen. Wir hätten keine Informationen darüber, wieso etwas häufig ist. In dem konkreten Fall der IP-Adressen wäre das schlecht. Das gesamte Teilnetz würde als gefährlich eingestuft, sobald 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 wird deshalb folgende Bedingung für das Zählen eingeführt: Nur konkrete Elemente einer Hierarchiestufe werden gezählt, die weder selbst häufig sind, noch zu einer anderen häufigen Hierarchiestufe weiter unten beitragen.

Komplexer wird das Vorgehen, wenn man zusätzlich noch mehrere Dimensionen beachten muss - also auch noch die Empfängeradresse wichtig ist. Wird die Häufigkeit

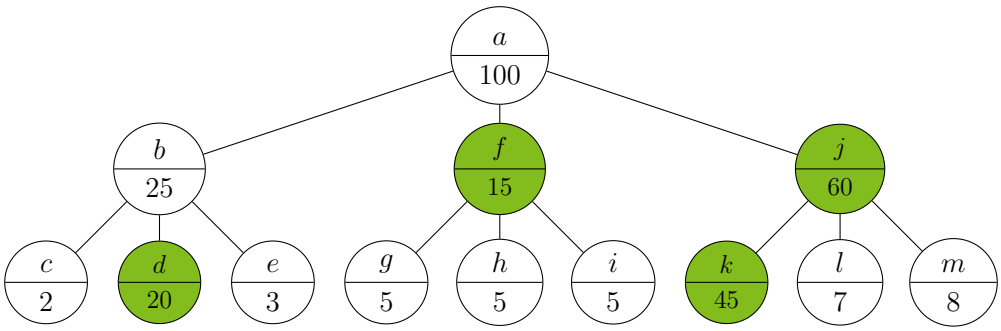


Abbildung 24: Hierarchical Heavy Hitters (HHH) bei einem absoluten Schwellwert von 10. Grüne Knoten sind HHH. d und l sind HHH, da sie direkt den Schwellwert überschreiten. Die Summe der nicht-HHH-Kinder von f (15) macht f auch zum HHH. j ist wegen m und n ein HHH, jedoch nicht wegen l. a und b sind keine HHH, da die Summe der nicht HHH Kinder nur 5 ist (c und e)

6 von $(a, 1)$ bei $(a, *)$ oder bei $(*, 1)$ hochgezählt (siehe Abbildung 25)? Hier unterscheidet man zwei Fälle:

split-case Die Häufigkeit wird zu allen Elternknoten propagiert

overlap-case Die Häufigkeit wird aufgeteilt, z.B. 50:50

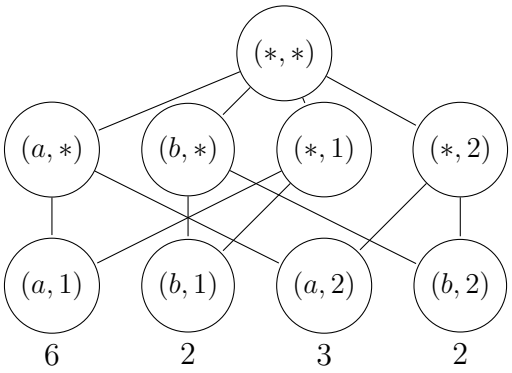


Abbildung 25: Mehrdimensionale Hierarchical Heavy Hitters - Jeder Knoten hat zwei Attribute. Das erste kann die Werte a oder b annehmen und das zweite 1 und 2.

Verwendet man z.B. die *overlap*-Regel ist dabei aber im Algorithmus zu beachten, dass einige Knoten eine viel zu große Häufigkeit erhalten. Angenommen die Häufigkeiten in Abbildung 25 wären bei den Blättern überall 1. Wäre der Schwellwert fünf, wäre der Wurzelknoten $(*, *)$ trotzdem häufig, obwohl nur insgesamt vier Elemente im Datenstrom aufgetaucht sind. Um dieses Problem zu vermeiden, muss man im Algorithmus alternierend zählen. Das bedeutet, dass man bei den direkten Eltern

addiert und bei den Eltern der Eltern subtrahiert. Dadurch wird das Problem des Zuvielzählens vermieden. Bei der *split*-Regel haben wir dieses Problem zwar nicht, dort wird bei den direkten Eltern aber zu wenig gezählt. Verwendet wird deshalb der *overlap*-Fall mit dem alternierenden Zählen.

HHH Algorithmen 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. Dies würde jedoch zuviel Speicher benötigen, was vor allem daran liegt, dass hier auch naiv alle Heavy Hitters gefunden werden. Die Differenzierung anhand der Hierarchical Heavy Hitters Bedingung erfolgt hier noch nicht. Hierfür muss man weitere Algorithmen benutzen, die im Grunde den Lossy Counting Algorithmus verwenden, zusätzlich aber auch Informationen zwischen den einzelnen Hierarchiestufen austauschen, um so die Bedingungen einzuhalten und möglichst wenig Speicher zu verwenden. Beispielsweise stutzt der *Full Ancestry* Algorithmus ganze Teilbäume im Hierarchiebaum, wenn diese nicht häufig genug sind. Nicht häufig bedeutet hierbei auch, dass die enthaltenen Elemente nicht mindestens in jedem Fenster aufgetaucht sind. Wird ein Teilbaum gestutzt, dann wird die bisherige Häufigkeit an den Elter propagiert. Dieses Stutzen bildet einen sogenannten *fringe* im Hierarchiebaum, der garantiert, dass unter dieser Grenze keine HHHs existieren. Über dieser Grenze wird aber auf jeden Fall richtig gezählt.

7.2 Lernalgorithmen auf verteilten Streams

7.2.1 Geometric Approach to Monitoring Threshold Functions over Distributes Data Streams

Es sei zu erwähnen, dass es sich bei diesem Verfahren um eine allgemeinen Ansatz handelt, der nicht nur auf den Fall der verteilten Spam-Erkennung beschränkt ist. Da der Anwendungsfall (siehe Kapitel 5.2) sich mit diesem Problem beschäftigt, soll im Folgenden der Algorithmus anhand dieses Beispiels vorstellt werden.

Ein ausführliches Beispiel Ausgehend von dem Beispiel welches in [21] beschrieben ist, folgt jetzt ein Überblick über alle benötigten Informationen.
Gegeben seien: n Agenten auf den n verschiedenen Emailservern $p_1, p_2, ..., p_n$.
$M_i = \{m_{i,1}, m_{i,2}, ..., m_{i,k}\}$ seien die letzten k Emails, empfangen am Emailserver p_i.
Sei $M = \bigcup_{i=1}^{n} M_i$ die Vereinigungsmenge der letzten k Emails, empfangen an jedem der n Emailserver.
Sei X eine Menge von Emails, dann bezeichnet: $Spam(X)$ die Emails, die als Spam markiert sind und $\overline{Spam(X)}$ die nicht als Spam markierten Emails.
Sei $Cont(X, f)$ die Menge von Email aus X, die Attribut f enthält und $\overline{Cont(X, f)}$ die

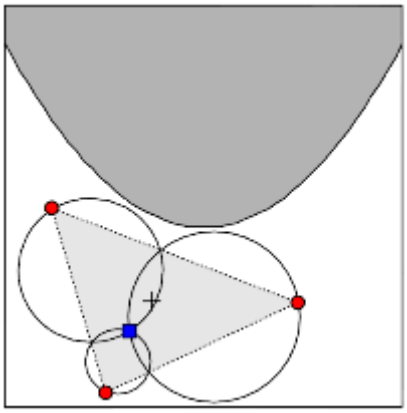


Abbildung 26:

Menge, die das Attribut f nicht enthält.
Sei $C_{f,X} = c_{i,j}$ eine 2×2 Kontingenztafel(Matrix) für das Attribut f über einer Menge von Emails X
$C_{1,1} = \frac{|Cont(X,f) \cap Spam(X)|}{|X|}$, $C_{1,2} = \frac{|Cont(X,f) \cap \overline{Spam(X)}|}{|X|}$,
$C_{2,1} = \frac{|\overline{Cont(X,f)} \cap Spam(X)|}{|X|}$, $C_{2,2} = \frac{|\overline{Cont(X,f)} \cap \overline{Spam(X)}|}{|X|}$.

Für die Knoten p_i ist C_{f,M_i} die lokale Kontingenztafel, während $C_{f,M} = \frac{\sum_{i=1}^{n} C_{f,M_i}}{n}$ die globale Kontingenztafel darstellt.
Von Interesse ist nun das Folgende: Für jedes Attribut f soll bestimmt werden, ob der Informationsgehalt der globalen Kontingenztafel über einem bestimmten Schwellwert r liegt oder nicht.
Der Informationsgehalt wird wie folgt bestimmt:
$G(C_{f,X}) = \sum_{i \in \{1,2\}} \sum_{j \in \{1,2\}} c_{i,j} \cdot log \frac{c_{i,j}}{(c_{i,1}+c_{i,2}) \cdot (c_{1,j}+c_{2,j})}$
Zusätzlich gilt: falls $c_{i,j} = 0$, dann folgt daraus, dass der gesamte obere Term mit 0 definiert ist.

Das Berechnungsmodel:
Sei $S = \{s_1, s_2, ..., s_n\}$ eine Menge von Streams der Knoten $P = \{p_1, p_2, ..., p_n\}$
Sei $\vec{v_1}(t), \vec{v_2}(t), ..., \vec{v_n}(t)$ ein d-dimensionaler Vektor, welcher aus den Streams abgeleitet wird. Die Werte der Vektoren ändern sich mit der Zeit t. Das sind die sog. lokalen Statistikvektoren.

In Abbilung 26 kann man dies gut beobachten. Die roten Punkte kennzeichnen dabei die Emailserver bzw. dessen lokale Statistikvektoren. Der blaue Punkt in der Mitte stellt den globalen Vektor dar. Im oberen Teil der Abbildung sieht man die Schwellwertfunktion, welche noch nicht überschritten ist (graue Fläche).
Seien $w_1, w_2, ..., w_n$ positive Gewichte, die den Streams zugeordnet wurden.

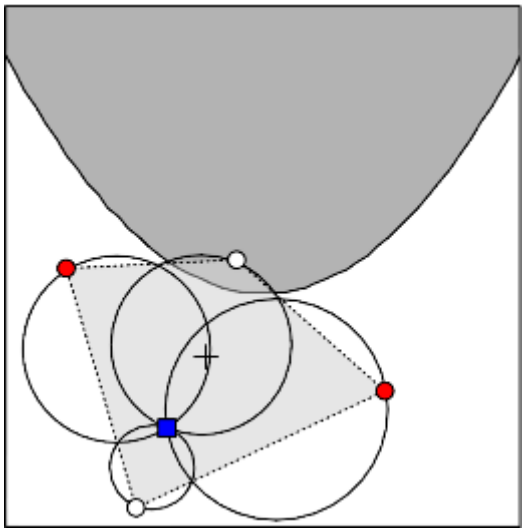


Abbildung 27: Schwellwert wurde übertreten

Bei den Betrachtung wird von einem sogenannten Sliding Window der Größe N_i ausgegangen. Das bedeutet, dass in jedem Knoten die Häufigkeit des Auftretens eines Elements gespeichert wird. Desweiteren sind alle Gewichte fix und allen Knoten bekannt. Dies kann später noch variieren.
Sei $\vec{v}(t) = \frac{\sum_{i=1}^{n} w_i \vec{v}_i(t)}{\sum_{i=1}^{n} w_i} \vec{v}(t)$ der sogenannten globale statistische Vektor. Sei weiterhin $f : \mathbb{R}^d \to \mathbb{R}$ eine beliebige Funktion, die sogenannten Überwachungs-/Kontrollfunktion. Wir wollen zu jedem Zeitpunkt t wissen, ob $f(\vec{v}(t)) > r$ gilt oder nicht. r ist dabei wieder der vordefinierte Schwellwert.
Abbildung 27 zeigt deutlich, dass der Schwellwert übertreten wurde. Diese Übertretung findet zuerst nur lokal im oberen Knoten(Emailserver) statt. Die anderen drei Knoten wissen von diesem Ereignis noch nichts und werden erst im Anschluss benachrichtigt.
Als nächstes wird ein Wahrscheinlichkeitsverteilungsvektor $\vec{e}(t)$ erstellt. Dieser wird aus den lokalen Statistikvektoren erzeugt, welche zu gewissen Zeitpunkten Daten von den Knoten erhalten. Mit $\vec{v'}_i$ wird der letzte statistische Vektor des Knotens p_i bezeichnet. Des Weiteren kennt jeder Knoten seinen letzten statistischen Vektor. Somit ist der Wahrscheinlichkeitsverteilungsvektor das gewichtete Mittel der letzten statistischen Vektoren der einzelnen Knoten: $\vec{e}(t) = \frac{\sum_{i=1}^{n} w_i \vec{v'}_i}{\sum_{i=1}^{n} \vec{w_i}}$. Von Zeit zu Zeit findet eine Aktualisierung des statistischen Vektors an einem oder an mehreren Knoten statt. Daraufhin wird der Wahrscheinlichkeitsverteilungsvektor aktualisiert. Zu jeder Zeit kennt jeder Knoten diesen Wahrscheinlichkeitsverteilungsvektor. Die Aktualisierung des statistischen Vektors geschieht durch ein Broadcast, bei dem der Knoten diesen

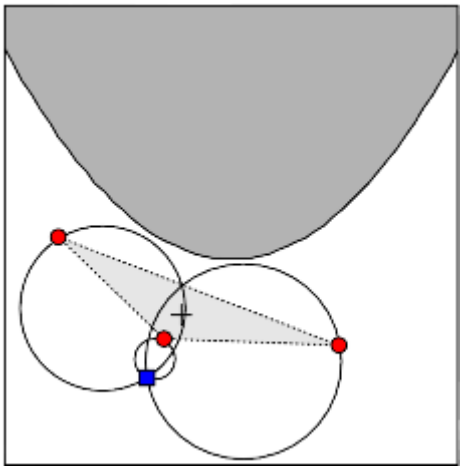


Abbildung 28: Rekonfiguration nach Schwellwertübertretung

an alle anderen Knoten sendet. Jeder Knoten merkt sich den letzten Broadcast der anderen Knoten und berechnet lokal auf dieser Basis den Wahrscheinlichkeitsverteilungsvektor.
Jeder Knoten p_i führt einen weiteren Parameter mit sich. Dies ist der sogenannten statistische Delta Vektor $\Delta\vec{v}_i(t)$. Mit diesem Vektor wird die Differenz zwischen dem derzeitigen lokalen statistischen Vektor und dem zuletzt berechneten beschrieben. Formal bedeutet das: $\Delta\vec{v}_i(t) = \vec{v}_i(t) - \vec{v'}_i$. Außerdem hat jeder Knoten zusätzlich noch einen anderen Parameter, den sogenannten Drift Vektor: $\vec{u}_i(t)$. Dieser wird in diesem Fall wie folgt bestimmt(dieser Wert wird für den dezentralen Fall anders berechnet!): $\vec{u}_i(t) = \vec{e}(t) + \Delta\vec{v}_i(t)$. Er beschreibt den Abstand von $\Delta\vec{v}_i(t)$ in Relation zum Wahrscheinlichkeitsverteilungsvektor.
Da bei diesem Ansatz die Knoten per Broadcast miteinander kommunizieren, kann es im Worst-Case passieren, dass für ein Versenden einer Nachricht zu n anderen Knoten n Punk-zu-Punkt Nachrichten benötigt werden. Bei einer Übertretung des Schwellwertes wird das Ereignis an die anderen Knoten kommuniziert, woraufhin eine Anpassung der lokalen Knoten stattfindet. In Abbildung 28 sieht man, wie sich die lokalen Knoten an die neue Situation anpassen.

7.2.2 Catching Elephants with Mice

In dem Artikel „Catching Elephants with Mice“[14] beschreiben die Autoren, wie große Sensornetzwerke auch durch eine spärliche Anzahl an Sensoren überwacht werden können. Vorraussetzung hierfür ist, dass eine geschickte Auswahl von zu überwachenden Knoten getroffen wird. Des Weiteren gilt die These nur unter der einschränkenden Annahme, dass das zu erkennende Ereignis geometrisch eine gebundene VC-Dimension (Vapnik-Chervonenkis-Dimension) aufweist. Unter Beachtung

dieser Einschränkungen erbringen die Autoren mit Hilfe statistischer Methoden den Beweis, dass in einem Netzwerk mit n beliebig verteilten Sensorknoten (*elephants*) die Auswahl von $O(\frac{1}{\epsilon}log\frac{1}{\epsilon})$ Sensorknoten (*mice*) ($0 < \epsilon < 1$) genügt, um alle Ereignisse, die $\Omega(\epsilon n)$ Knoten betreffen zu erkennen. Darüber hinaus kann die Größe des Ereignisses mit einem Schätzfehler von $\pm\frac{\epsilon * n}{4}$ vorhergesagt werden. Dieser Schätzfehler kann durch den Einsatz zusätzlicher Sensoren weiter verringert werden.

Zum besseren Verständnis des Problems bietet es sich an, dieses anhand eines Beispiels näher zu betrachten.

Praktisches Beispiel:
Ein Waldgebiet ist mit n Sensoren zur Erkennung von Waldbränden ausgestattet. Wir können annehmen, dass Waldbrände lokale Events mit einer gebunden VC-Dimension darstellen: sie treten zwar lokal auf, betreffen aber ab einer gewissen Größe definitiv eine erhöhte Anzahl benachbarter Sensorknoten.

8 Ausblick auf das zweite Semester

Die Kernaufgaben des zweiten Semesters wurden bereits in Kapitel 2.4 kurz erläutert. Im zweiten Halbjahr wird sich die Projektgruppe mit der Entwicklung weiterer Streaming-Algorithmen beschäftigen und außerdem eine Möglichkeit zur Evaluierung der Algorithmen bereit stellen. Im ersten Halbjahr wurden bereits zwei verschiedene Streaming-Algorithmen entwickelt um die beiden ausgewählten Anwendungsfälle umzusetzen. Um die Funktionalitäten des Frameworks auszubauen, soll die Anzahl der Streaming-Algorithmen aufbauend auf das im Wintersemester entworfene Framework ausgebaut werden.

Auch die Evaluierung der Algorithmen soll im zweiten Semester Thema sein. Dabei sollen zwei verschiedene Ansätze verfolgt werden. Zum einen soll es möglich werden gelabelte Testdaten durch den Algorithmus klassifizieren zu lassen und danach das gegebene mit dem generierten Label zu vergleichen. Mit Hilfe dieser Methode lässt sich zwar die Güte der Algorithmen prüfen, die Notwendigkeit der gelabelten Testdaten schränkt ihre Funktionalität aber stark ein. Daher soll in einem zweiten Ansatz auch die Möglichkeit geschaffen werden, entwickelte Streammining-Algorithmen mit *äquivalenten* Datamining-Algorithmen zu vergleichen. In diesem Fall könnte eine Evaluierung auch mit ungelabelten Daten durchgeführt werden, so lange die Güte des Datamining-Algorithmus bekannt ist.

Genau wie zu Beginn des Wintersemesters wird es auch zum Start des zweiten Halbjahres eine Seminarphase im Universitätskolleg Bommerholz geben. Die Vorträge der zweiten Seminarphase werden sich mit den oben genannten Themen auseinandersetzen, sind zum jetzigen Zeitpunkt aber noch nicht bekannt.

Anhang A : JavaDoc

9 Literatur

Literatur

[1] TOP 10 2007 - OWASP. http://www.owasp.org/index.php?title=Top_10_2007, 2007. [Online; Stand 6. Januar 2010].

[2] Die zehn größten Schwachstellen in Web-Anwendungen - TecChannel.de. http://www.tecchannel.de/webtechnik/webserver/2019842/schwachstellen_fehler_und_bugs_in_web_anwendungen/, 2010. [Online; Stand 6. Januar 2010].

[3] Hacker Manifest. http://www.hacker-ethik.de/hacker_manifest.htm, 2010. [Online; Stand 6. Januar 2010].

[4] Hackerkultur Hacker-Ethik. http://www.hacker-ethik.de/hackerkultur_hacker-ethik.htm, 2010. [Online; Stand 6. Januar 2010].

[5] 'Hacktivism' auf dem Vormarsch. http://www.tecchannel.de/sicherheit/news/1757637/, 2010. [Online; Stand 6. Januar 2010].

[6] BANERJEE, S., C. GROSAN und A. ABRAHAM: IDEAS: Intrusion Detection based on Emotional Ants for Sensors. In: ISDA, S. 344–349, 2005.

[7] BANERJEE, S., C. GROSAN, A. ABRAHAM und P. MAHANTI: Intrusion Detection on Sensor Networks Using Emotional Ants. Int'l J. Applied Science and Computations, 12(3):152–173, 2005.

[8] BRÖHL, A. (Hrsg.): Das V-Modell. Software - Anwendungsentwicklung - Informationssysteme. Oldenbourg, München [u.a.], 1993.

[9] BURGES, C. J. C.: A Tutorial on Support Vector Machines for Pattern Recognition. Data Min. Knowl. Discov., 2(2):121–167, 1998.

[10] CAUWENBERGHS, G. und T. POGGIO: Incremental and Decremental Support Vector Machine Learning. In: NIPS, S. 409–415, 2000.

[11] 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.

[12] 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.

[13] 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.

[14] GANDHI, S., S. SURI und E. WELZL: Catching elephants with mice: sparse sampling for monitoring sensor networks. In: SenSys '07: Proceedings of the 5th international conference on Embedded networked sensor systems, S. 261–274, New York, NY, USA, 2007. ACM.

[15] GLOGER, B.: Scrum: Produkte zuverlässig und schnell entwickeln. Hanser, München, 2. Aufl., 2009.

[16] INFORMATIONSTECHNIK, B. FÜR SICHERHEIT IN DER: Durchführungskonzept für Penetrationstests. https://www.bsi.bund.de/cln_164/ContentBSI/Publikationen/studien/pentest/index_htm.html. [Online; Stand 6. Januar 2010].

[17] LEE, W., S. J. STOLFO und K. W. MOK: A Data Mining Framework for Building Intrusion Detection Models. In: In IEEE Symposium on Security and Privacy, S. 120–132, 1999.

[18] MARKOWETZ, F.: Klassifikation mit Support Vector Machines. lecture notes "Genomische Datenanalyse", 2003.

[19] MORIK, K.: Stützvektormethode (SVM). lecture notes "Wissensentdeckung in Datenbanken", 2008.

[20] RAMOS, V. und A. ABRAHAM: ANTIDS: Self Organized Ant-Based Clustering Model for Intrusion Detection System. In: WSTST, S. 977–986, 2005.

[21] SHARFMAN, I., A. SCHUSTER und D. KEREN: A geometric approach to monitoring threshold functions over distributed data streams. ACM Trans. Database Syst., 32(4):23, 2007.

[22] VAPNIK, V. N.: Theorie der Zeichenerkennung. Akademie-Verlag, 1979.

[23] VAPNIK, V. N.: The Nature of Statistical Learning Theory (Information Science and Statistics). Springer, 1999.

[24] 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.