

# Projektbericht

---

## TMetrics

Projektseminar zum Thema Data Mining

---

### Autoren

Daniel Günther

Olaf Markus Köhler

Erwin Quiring

Björn Roß

Torsten Scholz

Wladimir Haffner

Sebastian Lichtenfels

Andreas Riddering

Jens Sandmann

Tobias Wenzel

### Betreuer

Prof. Dr. Jan Vahrenhold

Wolfgang Paul



# Inhaltsverzeichnis

<b>1. Einleitung</b>	<b>1</b>
1.1. Motivation . . . . .	1
1.2. Umgesetztes System . . . . .	2
<b>2. Projektorganisation</b>	<b>7</b>
2.1. Ideenfindung . . . . .	7
2.2. Scrum . . . . .	7
2.2.1. Grundmerkmale . . . . .	8
2.2.2. Rollen . . . . .	9
2.2.3. Verwendete Methoden . . . . .	11
2.2.4. Scrum, But . . . . .	14
2.3. Technologien . . . . .	16
2.4. Code-Organisation . . . . .	17
2.5. Tests . . . . .	18
<b>3. Architektur</b>	<b>20</b>
3.1. Beschaffung der Daten . . . . .	20
3.2. Darstellung der Daten . . . . .	22
3.3. Analyse der Daten . . . . .	24
<b>4. Infrastruktur</b>	<b>26</b>
4.1. Daemon . . . . .	26
4.1.1. Twitter-API . . . . .	26
4.1.2. Suchstrategie . . . . .	28
4.1.3. Parallele Suche . . . . .	37
4.1.4. Zeitnahe Ergebnisse . . . . .	43
4.1.5. Diskussion . . . . .	47

4.2.	Datenbank . . . . .	52
4.2.1.	Schema . . . . .	52
4.2.2.	Performance . . . . .	55
4.3.	REST . . . . .	58
4.3.1.	Einführung . . . . .	58
4.3.2.	Implementierung . . . . .	59
4.3.3.	Struktur des REST-Service . . . . .	62
4.3.4.	Umsetzung und Problemstellungen . . . . .	65
4.4.	Server . . . . .	70
4.4.1.	Anforderungen und Grundsteine . . . . .	70
4.4.2.	Projektname und Grundkonfiguration . . . . .	71
4.4.3.	Ubuntu-Update auf virtuellem System . . . . .	74
4.4.4.	Performance-Optimierungen . . . . .	75
<b>5.</b>	<b>Komponenten</b>	<b>77</b>
5.1.	Sentiment . . . . .	77
5.1.1.	Theoretische Grundlagen . . . . .	77
5.1.2.	Implementierte Klassifikatoren . . . . .	81
5.1.3.	Einbindung in die Architektur . . . . .	85
5.1.4.	Diskussion . . . . .	87
5.2.	Cluster-Analyse . . . . .	90
5.2.1.	Cluster-Analyse von Tweets . . . . .	92
5.2.2.	Cluster-Analyse von Hashtags . . . . .	95
5.2.3.	Verwendung der Hashtag-Cluster zur Tweet-Gruppierung . . . . .	98
5.2.4.	Cluster-Algorithmen . . . . .	98
5.2.5.	Multidimensionale Skalierung . . . . .	100
5.2.6.	Diskussion . . . . .	106
5.3.	Nachrichtenmodul . . . . .	109
5.3.1.	Identifikation interessanter Zeitpunkte . . . . .	109
5.3.2.	Sammeln passender Nachrichten . . . . .	111
5.4.	Frontend . . . . .	114
5.4.1.	Design . . . . .	114
5.4.2.	Verwendete Technologien . . . . .	117
5.4.3.	Datenhaltung . . . . .	118

5.4.4. Connection Pools . . . . .	120
5.4.5. Besonderheiten . . . . .	124
5.4.6. Diskussion . . . . .	134
<b>6. Ausblick und Fazit</b>	<b>136</b>
<b>Literaturverzeichnis</b>	<b>139</b>
<b>A. REST API</b>	<b>143</b>

# 1. | Einleitung

## 1.1. Motivation

Bei den Themenbereichen Big Data und Data Mining handelt es sich um Teilgebiete der Informatik, die in den vergangenen Jahren stark an Bedeutung gewonnen haben und daher aktuell von hohem Interesse sind. Es geht dabei um die Beschaffung, Analyse und Interpretation großer Datenmengen zur Gewinnung zusätzlicher Informationen. Aufgrund der Bedeutung und Aktualität dieser Gebiete war dementsprechend auch die Motivation der Projektteilnehmer groß, aktiv in diesem Bereich zu arbeiten und teilweise bereits vorhandene theoretische Kenntnisse in die Praxis umzusetzen. Ein ebenso bedeutsames Anwendungsgebiet dafür sind soziale Netzwerke, welche zudem den Vorteil besitzen, große Datenmengen öffentlich zur Verfügung zu stellen. Unsere Wahl fiel daher auf den Kurznachrichtendienst Twitter.

Damit war eine sehr allgemeine Projektidee unter der Überschrift „Data Mining auf Twitter“ gefunden. Um jedoch ein tatsächliches Projekt in der zur Verfügung stehenden Zeit umsetzen zu können, musste diese Idee auf einen spezifischen Anwendungsbereich konkretisiert werden. Vor dem offiziellen Beginn des Projekts bestanden dazu bereits von unterschiedlichen Projektmitgliedern eingebrachte Vorschläge. Diese basierten einerseits auf der Erkenntnis, dass sich die Aktivität auf Twitter möglicherweise mit der Bedeutung oder Aktualität eines bestimmten Themas in Verbindung setzen lässt, während der Inhalt der geposteten Nachrichten (auch Tweets genannt) einen Rückschluss auf die allgemeine Stimmung zu einem diskutierten Thema zulässt. Dieser Zugang erschien uns sinnvoll, da es interessant ist, ob sich aus den vorhandenen Informationen ein aussagekräftiges und realistisches Meinungsbild gewinnen lässt. Außerdem erfordert eine solche Analyse den Einsatz von Verfahren aus dem *Machine Learning*, ein weiteres aktuelles Gebiet, in dem sich wertvolle Erfahrungen sammeln lassen. Auf der anderen Seite ließ sich aus unserer Sicht die Tatsache nutzen, dass Tweets eine räumliche und zeitliche Dimension aufweisen, das heißt es ist bekannt, wo und wann ein Tweet erstellt wurde. Das eröffnet die

Möglichkeit, die Popularität eines Themas zeitlich nachzuvollziehen oder seine räumliche Ausbreitung darzustellen. Abhängig von der Komplexität dieser räumlichen Analyse ließen sich hier auch Verfahren aus der algorithmischen Geometrie einbringen. Als eine anfängliche praktische Anwendung dieser Betrachtungsweisen wurde zunächst die Politik vorgeschlagen. Das stand unter dem Eindruck gesellschaftlicher Entwicklungen, in denen Twitter eine Rolle spielte, wie z.B. des arabischen Frühlings und ähnlicher Protestbewegungen. Die Verbreitungen solcher Entwicklungen lassen sich beispielsweise räumlich nachvollziehen. Als Anwendung des Meinungsbildes war vorgesehen, eine Aussage über den Ausgang von Wahlen treffen zu können.

Für die Organisation der Umsetzung dieser Ideen wurden Verfahren der agilen Software-Entwicklung eingesetzt. Dies sollte den Projektmitgliedern ermöglichen, Erfahrungen mit den damit verbundenen Vorgehensweisen und Prinzipien wie beispielsweise Scrum zu sammeln und dem Projekt zusätzlich zum akademischen auch einen praktischen Charakter verleihen. Das ist besonders unter dem Hinblick von Bedeutung, dass sich die Anforderungen im Projektverlauf ständig verändern können und das Team in der Lage sein muss, darauf zu reagieren.

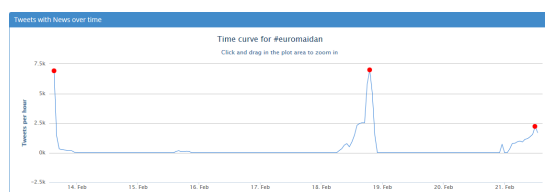
## 1.2. Umgesetztes System

Aus diesem Prozess ging schlussendlich der finale Zustand des Systems hervor. Wie bereits erläutert, war die Grundidee, zu einem spezifischen Thema oder Begriff die Daten aus Twitter aus einer Reihe verschiedener Gesichtspunkte darzustellen. Das System sollte dabei als Anwendung im Browser realisiert werden, wobei wir uns im Hinblick auf Eleganz und Bedienbarkeit an Seiten wie Wolfram Alpha<sup>1</sup> orientiert haben. Aus diesem Grund besteht die Startseite der Anwendung nur aus einem Eingabefeld für den gewünschten Suchbegriff.

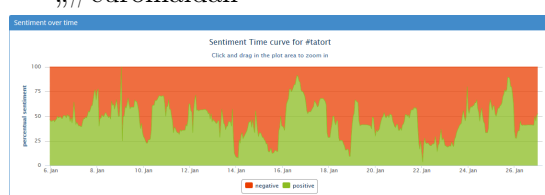
Ist ein Suchbegriff eingegeben, beginnt die Analyse. Dabei wird unterschieden, ob in unserer Datenbank bereits Daten für diesen Begriff vorliegen oder nicht. Falls noch keine Daten vorhanden sind, wird der Suchbegriff in die Datenbank eingetragen und es wird damit begonnen, Daten zu diesem Suchbegriff von Twitter abzurufen und abzuspeichern. Wenn dann Daten vorhanden sind, werden die Daten von unserem System analysiert und anschließend dem Benutzer angezeigt. Wie bereits erwähnt, liegen die Ergebnisse nach spezifischen Gesichtspunkten vor, welche jeweils in Boxen dargestellt werden, die

---

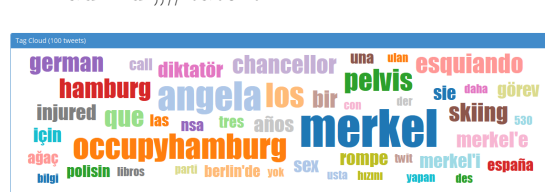
<sup>1</sup>[www.wolframalpha.com](http://www.wolframalpha.com)



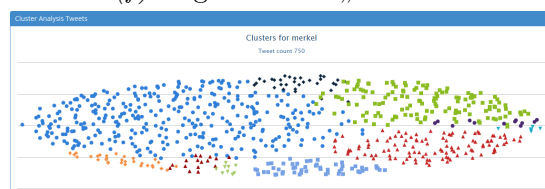
(b) Anzahl Tweets im zeitlichen Verlauf zu „#euromaidan“



(d) Relatives Sentiment im zeitlichen Verlauf zu „#tator“



(f) Tag Cloud zu „Merkel“



(h) Cluster-Analyse von Tweets zu „Merkel“

timentanalyse“ (1.1c) betrachtet, bei der jedem Tweet eine positive, negative oder neutrale Stimmung (Sentiment) zugewiesen ist. Die zeitliche Ausbreitung ist in der Ansicht „Tweets im zeitlichen Verlauf“ (1.1b) zu sehen, welche die Anzahl Tweets zum Suchbegriff pro Stunde auf einem Graphen aufträgt. Eine Kombination aus Stimmungsbild und zeitlicher Ausbreitung wird in der Ansicht „Relatives Sentiment im zeitlichen Verlauf“ (1.1d) angeboten. Das Clustering von Tweets gemäß ihrer Gemeinsamkeiten ist in der Ansicht „Cluster-Analyse von Tweets“ (1.1h) umgesetzt worden. Eine Variation dieses Ansatzes findet sich in der „Cluster-Analyse von Hashtags“ (1.1g), bei der – wie der Name schon vermuten lässt – an Stelle von einzelnen Tweets Hashtags zu einem Suchbegriff zu Clustern zusammengefasst werden. Die dort auftauchenden Hashtags sind allesamt Tweets zu diesem Suchbegriff entnommen. Eine Übersicht über die häufigsten dieser Hashtags bietet die Ansicht „Verwandte Hashtags“ (1.1e), welche die zehn meistbenutzten Hashtags zum eingegeben Suchbegriff auflistet. Eine Verallgemeinerung dieser Ansicht ist die „Tag Cloud“ (1.1f), in der nicht nur Hashtags, sondern generell alle Worte (bis auf Worte ohne eigentlichen Informationsgehalt, sogenannte *stop words*) ihrer Häufigkeit entsprechend angezeigt werden. Dabei wird die Häufigkeit eines Wortes durch die Größe der Darstellung repräsentiert: Je größer ein Wort dargestellt wird, desto häufiger ist es in Tweets zum analysierten Suchbegriff vorhanden. Außerdem existiert noch die „Sprachverteilung“ (1.1a), welche die absolute und relative Häufigkeit der Sprachen, in denen die Tweets eines Suchbegriffs verfasst sind, grafisch präsentiert.

Die Ergebnisse zu einem Suchbegriff werden in der Anzeige in einem Tab zusammengefasst, in dem die einzelnen Ansichten angezeigt werden. Bei der Suche nach weiteren Begriffen wird jeweils ein weiterer Tab hinzugefügt, der wiederum die einzelnen Ansichten enthält. Bei der Analyse von zwei oder mehr Suchbegriffen wird zudem ein weiterer Tab mit einer Vergleichsansicht aller eingegebenen Suchbegriffe präsentiert. Diese Vergleichsansicht enthält eine Ansicht zum Vergleich der Stimmungsbilder und eine Ansicht zum Vergleich der zeitlichen Verläufe der einzelnen Suchbegriffe, was in Abbildung 1.2 zu sehen ist.

Zusätzlich zur reinen Präsentation der analysierten Ergebnisse haben viele der soeben beschriebenen Ansichten auch Funktionalitäten, die exploratives Arbeiten mit dem System und den darin enthaltenen Daten ermöglicht. Beispielsweise ist es möglich, die zu analysierenden Daten zu einem Suchbegriff nach Sprachen zu filtern. Zusätzlich zum Sprachfilter in der Navigationsleiste ist diese Funktionalität auch aus der Ansicht „Sprachverteilung“ verfügbar. Ebenso lässt sich die Suche nach einem weiteren Begriff nicht nur



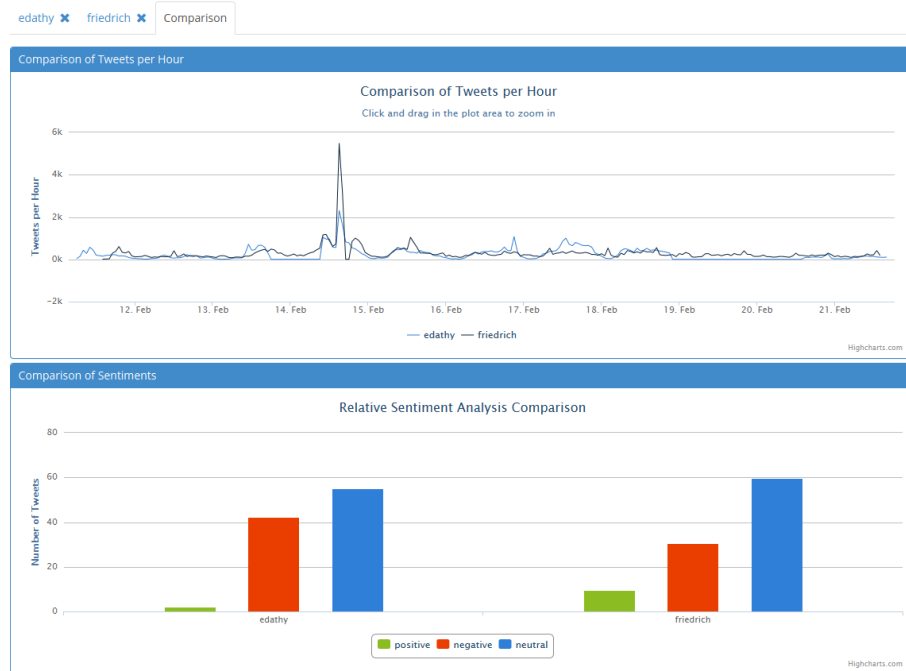


Abbildung 1.2.: Vergleichsansicht zu den Suchbegriffen „edathy“ und „friedrich“

durch dessen Eingabe in die Suchleiste starten, sondern auch durch einen Klick auf den gewünschten Suchbegriff in der „Tag Cloud“ oder bei „verwandten Hashtags“. Eine Möglichkeit, detaillierte Informationen zu erhalten, bietet die Anzeige einzelner Tweets in unserem System. In der Ansicht zur „Cluster-Analyse von Tweets“ ist dies durch einen Tooltip realisiert, in den Ansichten „Sentimentanalyse“ und „Tweets im zeitlichen Verlauf“ öffnet sich ein modaler Dialog mit einer detaillierten Anzeige von einzelnen Tweets.

Zusätzlich zur Anzeige einzelner Tweets bietet die Ansicht „Tweets im zeitlichen Verlauf“ noch die Anzeige von News zu diesem Suchbegriff. Diese werden aus einer zusätzlichen Suche der drei externen Quellen „Bing Web“, „Bing News“ und „Google News“ akquiriert. Dieses News-Modul wurde während des agilen Entwicklungsprozesses an Stelle des Kinomoduls umgesetzt, da eine Einbindung in das bis dahin bestehende System sowohl vom Product Owner, als auch von uns als sinnvoller erachtet wurde. Eine weitere der ursprünglichen Projektideen, die aufgrund der Priorisierung durch den Product Owner herausfiel, ist die Anzeige der räumlichen Verbreitung von Tweets. Der Hauptgrund hierfür ist die relativ geringe Anzahl an Tweets, zu denen eine Position vorhanden ist.

In den folgenden Kapiteln wird das hier kurz beschriebene System und dessen Entstehungsprozess im Detail erläutert. Dazu wird in Kapitel 2 zunächst beschrieben, wie

wir das Projekt intern organisiert haben. Dies betrifft sowohl die Organisation des Systems als auch die von uns verwendete Vorgehensweise. Im darauf folgenden Kapitel 3 wird die Architektur unseres Systems beschrieben, welches in die drei grobe Bereiche Beschaffung, Analyse und Darstellung von Daten aufgegliedert ist. Die daraus entstandene Infrastruktur wird in Kapitel 4 detailliert geschildert, woraufhin in Kapitel 5 Details zu den einzelnen Komponenten und Ansichten gegeben werden. Das abschließende Kapitel 6 fasst das zuvor Beschriebene kurz zusammen und bietet einen Ausblick auf mögliche weiterführende Ansätze, die auf dem von uns entwickelten System aufbauen.

## 2. | Projektorganisation

### 2.1. Ideenfindung

Wie eingangs beschrieben stand zu Beginn des Projekts noch ein weites Feld an Themen und Anwendungen zur Verfügung. Um in diesem Feld ein konkretes Projektziel zu identifizieren, das innerhalb eines Projektseminars erreicht werden kann, teilte sich das Team in Absprache mit den Projektbetreuern in fünf Gruppen auf, die jeweils einen geeigneten Vorschlag für das Projekt erarbeiten sollten. Bei der Abstimmung über die Möglichkeiten kristallisierten sich dabei zwei Favoriten heraus. Anstatt uns vollständig für einen dieser Vorschläge zu entscheiden, kombinierten wir diese Alternativen. Damit sieht das Projekt in der finalen Fassung vor, das Meinungsbild auf Twitter zu einem bestimmten Thema aus einer Reihe von Gesichtspunkten darstellen zu können. Von der einen Projektidee stammen die Gesichtspunkte des zeitlichen Verlaufs, der räumlichen Verbreitung und der aktuellen Stimmungslage. Das Clustering von Tweets gemäß ihrer Gemeinsamkeiten bildete die zentrale Idee des anderen Vorschlages und wurde als weiter Gesichtspunkt aufgenommen. Dies wurde um die Idee des ersten Vorschlags erweitert, Module anzubieten, welche die verfügbaren Daten in einem spezifischen Kontext interpretieren und dabei auch weitere externe Daten heranziehen. Als konkretes Modul sollte dabei ein sogenanntes „Kinomodul“ implementiert werden, welches Informationen aus externen Filmdatenbanken mit den Ergebnissen aus Twitter kombiniert, um Filme vergleichen zu können und Empfehlungen möglich zu machen.

### 2.2. Scrum

Wie bereits erwähnt, wurde für die Organisation des Projektes auf Hilfsmittel aus der agilen Softwareentwicklung zurückgegriffen. Im Gegensatz zur klassischen Softwareentwicklung z. B. nach dem Wasserfallmodell mit teils langen Phasen, sind die Hauptmerkmale der agilen Softwareentwicklung vor allem flexiblere und oftmals vor allem zeitlich weniger

aufwändige Methoden. Im Vordergrund stehen neben weiteren Merkmalen Schlagwörter, wie z. B. kurze Iterationen, testgetriebene Entwicklung oder häufige Refaktorisierungen. Die Anzahl der Methoden, die sich unter dem Stichwort agile Softwareentwicklung zusammenfassen lassen, ist relativ groß, sodass sich kaum sämtliche Methoden konsequent innerhalb eines Projektes umsetzen lassen. Es haben sich daher verschiedene Vorgehensmodelle, wie z. B. Extreme Programming, Kanban oder Scrum entwickelt. Da wir für die Projektorganisation die Methoden aus Scrum verwendet haben, möchten wir im Folgenden einen Überblick darüber geben, welche Methoden von uns angewandt wurden und auf welche wir aus welchen Gründen verzichten mussten.

### 2.2.1. Grundmerkmale

Ohne Scrum an dieser Stelle im Detail erklären zu wollen, seien einige wichtige Grundmerkmale genannt, welche sich auch in unserem Projektseminar wiedergefunden haben.

Im Rahmen von Scrum werden die Anforderungen zunächst grob als sogenannte User Stories formuliert und anschließend in möglichst spezifische einzelne Teilaspekte (sogenannte Arbeitspakete) unterteilt. Nachdem die Projektidee ausgereift ist, wird dies innerhalb des Teams im Sinne von Scrum umgesetzt. Anschließend werden die Aufwände der Arbeitspakete innerhalb des Teams geschätzt. Details hierzu folgen in einem der nächsten Abschnitte.

Da das Vorgehen über die komplette Entwicklungszeit nicht aus einzelnen langen und aufeinander folgenden Phasen (wie z. B. im Wasserfallmodell) besteht, sondern bei Scrum auf kurze ca. 2-3 Wochen dauernde und wiederholende Iterationen gesetzt wird, folgt nach der Aufwandschätzung das Aufteilen der vorhandenen User Stories auf die jeweils kommende Iteration. Das von Scrum erklärte und von uns in der Regel erreichte Ziel ist hierbei, am Ende einer jeden Iteration ein funktionsfähiges Stück Software in der Hand zu halten.

Ein weiteres erklärtes Grundprinzip von Scrum ist eine rege Kommunikation, gestützt durch häufige und verschiedene Arten von Meetings. Auch wir konnten im Rahmen des Projektseminars mit den verschiedenen Arten von Meetings unsere Erfahrungen sammeln.

In Kombination mit den kurzen Iterationen und dem Endziel einer jeden Iteration führt dies dazu, dass das Entwicklungsteam schnell auf (sich ggf. ändernde) Kundenwünsche reagieren kann und so mögliche und vor allem langfristig folgenreiche Missverständnisse in großen Teilen verhindert werden können. Auch auf unser Projektteam kamen im Rahmen

der Entwicklung einige neue Kundenwünsche zu, welche zumeist erfolgreich und zeitnah umgesetzt werden konnten.

Um den Erfolg von Scrum sicherzustellen, ist auf die Einhaltung dieser Grundprinzipien zu achten. Soweit dies im Rahmen einer Lehrveranstaltung möglich ist, wurde dies vom Projektteam auch gemacht, sodass man festhalten kann, dass Scrum für unser Projekt definitiv ein Erfolg war.

Um die Grundprinzipien und die Umsetzung innerhalb unseres Projektes ein wenig weiter zu beleuchten, werden wir im Folgenden auf einige Rollen und Methoden im Detail eingehen.

## 2.2.2. Rollen

### Scrum-Team

Innerhalb des Scrum-Modells gibt es verschiedene Rollen. Die Projektteammitglieder nahmen hierbei natürlich in erster Linie die Rolle der Entwickler ein. Das Entwicklerteam wird bei Scrum auch als Scrum-Team bezeichnet. Ein wichtiges Merkmal ist, dass dieses Team in großen Teilen selbstverwaltet arbeitet und eventuell auftretende Probleme intern zu lösen versucht. Dies hat bei uns sehr gut funktioniert. So wurden die verfügbaren Aufgaben ohne Komplikationen innerhalb des Teams verteilt und auch ein auftretendes Kommunikationsproblem konnte autark innerhalb des Teams gelöst werden.

### Product Owner

Die Rolle des Product Owners ist am ehesten mit der eines Projektleiters oder Firmeninhabers in einem klassischen Szenario zu vergleichen. Der Product Owner ist derjenige, der das fertige Produkt nach außen verkauft und mit dem Kunden in engem Kontakt steht. Letztendlich nimmt er daher immer auch die Rolle eines Vermittlers zwischen Kunden und Entwicklern ein, die ansonsten nicht unbedingt in direktem Kontakt miteinander stehen müssen.

In unserem Szenario wurde diese Rolle durch einen unserer Projektbetreuer besetzt. Er wohnte in dieser Rolle z. B. den sogenannten Sprint-Planning-Meetings bei und sortierte in diesen Meetings die gefundenen User Stories nach einer Priorität im Sinne des Kunden. Generell fanden im Rahmen des Projektseminars regelmäßige Meetings mit dem Product Owner statt, welche je nach aktueller Situation innerhalb der Iteration verschiedene Aus-

prägungen erhielten und somit nicht nur zum Sprint Planning wurden, sondern z. B. auch zum Sprint Review wurden.

### **Scrum Master**

Der Scrum Master ist mit dem Entwicklerteam eng verbunden, aber stellt eigentlich kein Mitglied des Entwicklerteams, sondern eine Art Bindeglied zum Product Owner dar. Der Scrum Master hält dem Entwicklerteam den Rücken frei und sorgt z. B. dafür, dass die benötigten externen Anforderungen erfüllt sind. Damit hat er vor allem zu Beginn eines Projektes einen etwas erhöhten organisatorischen Aufwand zu bewältigen und nimmt in der restlichen Zeit eine eher moderierende Rolle in den verschiedenen Meetings ein. Normalerweise wechselt im Verlaufe eines Projektes der Scrum Master nicht, sondern wird durchgehend durch die selbe Person repräsentiert. Damit alle Projektmitglieder aber einmal in die Position des Scrum Masters kommen konnten, mussten wir mit diesen Prinzipien leider brechen. Der Scrum Master war also bei uns abweichend von Scrum sowohl durch wechselnde Personen repräsentiert als auch stets ein Mitglied des Entwicklerteams. Das tat dem Moderieren der verschiedenen Meetings aber keinen großen Abbruch, sodass letztendlich alle Projektteammitglieder von den Einblicken in diese Rolle profitiert haben.

### **Kunde**

Als weitere Rolle gibt es natürlich den Kunden, welcher in unserem Szenario durch den anderen Projektbetreuer verkörpert wurde. Der Kunde stellt natürlich die anfänglichen grundsätzlichen Anforderungen an das Projekt und beauftragt das gesamte Team mit der Entwicklung. Obwohl die Projektidee in diesem Fall durch das Projektteam selbst entwickelt wurde, hat der Kunde es sich nicht nehmen lassen im Verlauf des Projektes ändernde Anforderungen einzubringen, wie z. B. dem Wunsch nach einem “Kachelinterface“. Im eigentlich Scrum-Sinne interagiert der Kunde nur bedingt mit dem gesamten Entwicklerteam, hiervon wurde im Rahmen des Projektseminars ebenfalls ein wenig abgewichen. Dies hat aber insgesamt zu keiner negativen Beeinflussung im Sinne von Scrum geführt und sich teilweise durch kurze Kommunikationswege eher als Vorteil herausgestellt.

## 2.2.3. Verwendete Methoden

### User Story

Die einzelnen Aufgabenkomplexe werden als User Stories bezeichnet, entsprechen also einer eher umgangssprachlichen Formulierung aus der Sicht des Benutzers, was in der Anwendung konkret passieren soll. Konkretisiert werden diese User Stories durch die Entwickler in dem sie auf kleinere konkrete Aufgaben, sogenannte Tasks unterteilt werden. User Stories als auch Tasks wurden von uns auf (unterschiedlich großen) Karteikarten festgehalten, um diese später flexibel dem Scrum-Board hinzufügen zu können. Details zum Scrum-Board folgen in einem späteren Abschnitt.

Folgende beispielhafte User Story wurde verfasst, um die Tag-Cloud-Anzeige zu implementierten:

2.4	Als Benutzer möchte ich Tag-Clouds haben
	Geschätzter Aufwand 5 Punkte
	Benötigte Stunden 29

In der Abbildung 1.1f ist die Implementierung im Frontend abgebildet. Für die Implementierung der Tag Cloud wird eine REST-Anfrage benötigt, die die benötigten Datenbank-anfragen macht, sowie die Visualisierung im Frontend. Deswegen wurde die User Story in folgende Tasks aufgebrochen:

2.4.1	REST-Anfrage implementieren						
	Aufwand 2 Punkte						
	<table> <tr> <th>Beteiligte Personen</th><th>Benötigte Stunden</th></tr> <tr> <td>Person1</td><td>2</td></tr> <tr> <td>Person2</td><td>4</td></tr> </table>	Beteiligte Personen	Benötigte Stunden	Person1	2	Person2	4
Beteiligte Personen	Benötigte Stunden						
Person1	2						
Person2	4						

2.4.2	Visualisierung im Frontend implementieren						
	Aufwand 3 Punkte						
	<table> <tr> <th>Beteiligte Personen</th><th>Benötigte Stunden</th></tr> <tr> <td>Person2</td><td>12</td></tr> <tr> <td>Person3</td><td>11</td></tr> </table>	Beteiligte Personen	Benötigte Stunden	Person2	12	Person3	11
Beteiligte Personen	Benötigte Stunden						
Person2	12						
Person3	11						

An der Nummerierung oben links erkennt man, dass die User Story und die Tasks zu der zweiten Iteration gehören, die User Story hat die Nummer vier, wobei die Nummerierung keine besondere Rolle spielt und lediglich der eindeutigen Identifizierung dient. Die Tasks haben entsprechend die Nummern eins und zwei.

Auf den User-Story-Karten wurde protokolliert, wie hoch der Aufwand in Punkten (hier 5) dieser Story ist, welcher sich durch die Summe des Aufwandes der einzelnen Tasks ergibt. In diesem Fall 2 Punkte für den Task 2.4.1 und 3 Punkte für den Task 2.4.2. Ebenfalls wurde an den User Stories notiert, wie hoch der benötigte Aufwand in Stunden tatsächlich ist. Dieser setzt sich wie die Punkte aus der Summe der benötigten Stunden für die einzelnen Tasks zusammen, welche von dem jeweiligen Entwickler nach Fertigstellung eines Tasks auf der Task-Karteikarte festgehalten wurde.

Durch diese Art der Buchführung über verbrauchte Punkte und Stunden, ist es relativ einfach möglich, den jeweils aktuellen Fortschritt innerhalb der Iteration zu kontrollieren und sich einen Überblick darüber zu verschaffen, ob am Ende der Iteration alle Tasks und Stories fertig sind oder ob noch etwas offen bleiben wird. Mehr dazu später im Abschnitt über den Burndownchart.

## Planning Poker

Im Gegensatz beispielsweise zum Wasserfallmodell wird bei Scrum der Aufwand nicht vom Projektleiter oder Geschäftsführer bestimmt und quasi „von oben“ diktiert, sondern der Aufwand für die Implementierung verschiedener Anforderungen wird aus dem Team heraus bewertet und geschätzt. Hierzu bedienen sich die beteiligten Entwickler einem Schätzverfahren, das unter dem Namen Planning Poker bekannt ist. Als Referenz-Aufwand wird eine im Team bekannte Aufgabe heran gezogen, anhand derer der Aufwand einer neuen Anforderung geschätzt werden soll. Der Hauptvorteil des Planning Pokers liegt darin, dass die beteiligten Personen sich zunächst nicht gegenseitig beeinflussen und daher trotz der allgemein bekannten Referenz-Aufgabe sehr abweichende Schätzungen entstehen können. Diese abweichenden Schätzungen eröffnen dann Diskussionen über den tatsächlichen Umfang und Aufwand der Aufgaben. Dies hatte auch bei uns im Projektteam die Folge, dass bei nicht-trivialen Aufgaben der Sachverstand aller Mitglieder einging und damit für jeden eine genauere Kenntnis der Anforderungen erreicht wurde.

Wie wir anhand der User Stories und einzelnen Tasks bereits gesehen haben, wird dieser Aufwand in Punkten geschätzt, wobei Punkte bei Scrum ausdrücklich nicht den Arbeitsstunden entsprechen, sondern als eine Art Einschätzung der Komplexität der Aufgabe zu



verstehen sind. Dies ist von Vorteil, wenn sich Entwickler mit unterschiedlichen Erfahrungswerten an die selbe Aufgabe machen und sicherlich unterschiedliche Zeit benötigen würden. Da der Punktestand gemessen werden kann, bietet dies einen besseren Überblick über den Fortschritt der Iteration.

Die Summe der geschätzten Teilaufgaben und User Stories landen gesammelt in dem Product Backlog, welches somit das Pendant zu dem sonst bekannten Pflichten- oder Lastenheft darstellt. Das Pendant zu dem Product Backlog war in unserem Fall die Sammlung der noch nicht geplanten User Stories, welche auf Karteikarten notiert wurden, um diese inklusive der zugehörigen Tasks, an unserem Scrum-Board anbringen zu können.

### **Scrum-Board**

Das Scrum-Board ist eine Art Tafel oder White-Board, welches an einer für alle Beteiligten gut sichtbaren Stelle aufgehängt wird. Es ist ein sehr essenzieller Bestandteil von Scrum und dient dazu, einen Überblick über den Fortschritt der einzelnen Aufgaben zu geben, sowie den Entwicklern die Möglichkeit zu geben sich flexibel noch nicht bearbeitete Aufgaben heraus zu nehmen. Aus der durch den Product Owner priorisierten Liste von User Stories aus dem Sprint Backlog, kann nun eine Story heraus gegriffen und an dem Scrum-Board angebracht werden. Dies signalisiert jedem, dass aktuell daran gearbeitet wird. Die einzelnen Arbeitspakete können aber dennoch von unterschiedlichen Entwicklern bearbeitet werden. Eine Idee von Scrum ist, dass sich ein Entwickler hier durchaus auch mal eine Aufgabe aus einem Gebiet nehmen kann, in dem er nicht unbedingt vollständig eingearbeitet ist, um so seinen eigenen Horizont zu erweitern und auch durch einen anderen Blickwinkel der Entwicklung dieser User Story evtl neue Impulse hinzuzufügen. Aufgrund steigender Komplexität der einzelnen Komponenten hat dieser Fachgebiets-Wechsel-Effekt im Laufe des Projektes nach und nach abgenommen, was jedoch durchaus zu erwarten war. Generell war der Überblick der offenen, geschlossenen und in Bearbeitung befindlichen Stories und Tasks jedoch sehr hilfreich.

### **Burndownchart**

Neben dem Scrum-Board ist das Burndownchart ein eben so wichtiger Bestandteil von Scrum. Immer wenn am Scrum-Board eine Aufgabe aus dem Bereich der in Bearbeitung befindlichen Stories und Tasks kennzeichnet, heraus genommen wird, wird der in Punkten auf der Story notierte Aufwand von der Summe der Aufwände der Iteration abge-

zogen und ein Restwert des Aufwandes wird ermittelt. Dieser Ist-Wert des Rest-Aufwands wird anschaulich an einer Kurve dargestellt, welche als Burndownchart bezeichnet wird. Idealerweise verläuft diese Kurve möglichst nah an einer gedachten (und je nach Ausprägung auch eingezeichneten) Ideallinie, welche diagonal von 100% des Aufwandes bis zu 0% über den zeitlichen Verlauf der Iteration verläuft. In der Praxis liegt diese Linie doch oftmals oberhalb dieser Ideallinie, da Stories erst nach Beendigung vollständig von dem Rest-Wert abgezogen werden und da durch Probleme die im Verlauf der Iteration auftreten, die Summe des Gesamtaufwandes oft noch ansteigen kann. Um dies besser zu visualisieren haben wir eine weitere Linie in unserem Burndownchart eingeführt, welche den 100% Wert des Aufwandes darstellt und an entsprechenden Stellen dann einen Knick nach oben bekommt. Somit konnte man sich dennoch einen guten Überblick darüber verschaffen, wie gut man mit den geplanten Aufgaben dieser Iteration im Zeitplan ist.

### Definition of Done und Testgetriebene Entwicklung

Wie im vorherigen Abschnitt beschrieben, konnte man unserem Scrum-Board stets entnehmen, welche Aufgaben bereits abgeschlossen sind. Damit sich zwischen den Mitgliedern keine unterschiedlichen Auffassungen etablieren, wann eine Aufgabe als abgeschlossen bezeichnet werden kann, sieht Scrum eine sogenannte *definition of done* (DoD) vor. Eine solche DoD wurde vom Team innerhalb der ersten Wochen erarbeitet und stellte seitdem ein verbindliches Regelwerk dar. Vor allem war ein fester Bestandteil der DoD, dass die jeweilige Komponente getestet worden sein soll, bevor die Aufgabe als abgeschlossen bezeichnet werden kann. Dies wurde durch automatisierte Tests erreicht, worüber in einem kommenden Abschnitt noch mehr zu erfahren sein wird.

### 2.2.4. Scrum, But

Da bislang die Scrum-Methoden betrachtet wurden, welche vollständig oder zumindest in großen Teilen nach Scrum-Vorgabe durch uns genutzt und umgesetzt wurden, möchten wir noch auf einige Aspekte eingehen, welche in unseren Augen nur teilweise bzw. mit einigen Abweichungen von uns genutzt wurden.

#### Daily Scrums

Diese Form von Meeting sollte wie der Name schon andeutet täglich stattfinden. Da sich aber kaum ein täglicher fixer Termin finden ließ, an dem stets alle oder zumindest

die meisten Teammitglieder hätten anwesend sein können, wurde innerhalb unseres Team hieraus ein D-Daily-Scrum gemacht, sodass wir uns lediglich Dienstags und Donnerstag zu dieser Art von Meeting zusammen gefunden haben. Inhaltlich geht es beim Daily Scrum eigentlich darum, dass jedes Mitglied kurz darlegt, was seit dem letzten Daily-Scrum geschehen ist, also letztendlich was entwickelt wurde und was bis zum nächsten Daily Scrum geschehen soll. Ebenfalls sollte kurz erwähnt werden, ob mögliche Komplikationen zu erkennen sind, welche das Ziel bis zum nächsten Daily Scrum gefährden könnten. Dies würde der Scrum Master dann zum Anlass nehmen, aktiv zu werden um die möglichen Hindernisse aus dem Weg zu räumen.

Um den Charakter eines Kurz-Meetings zu stärken und nicht in langwierige Diskussionen abzudriften, wird empfohlen, diese Art von Meetings im Stehen abzuhalten, möglichst auch außerhalb des regulären Arbeitsbereiches. Dies wurde vom Team auch umgesetzt, indem man sich im Erdgeschoss des Institutes traf und besagte Punkte ansprach. Vermutlich aufgrund der nicht täglich stattfindenden Daily-Scrum-Meetings neigten die Meetings teilweise zu längeren Diskussionen und übertrafen dann definitiv die reine Bestandsaufnahme.

### **Retrospektive und Sprint Review**

Aufgrund der eingeschränkten Anzahl an gemeinsamen Terminen pro Woche ließen sich die Retrospektiven und Sprint Reviews leider nicht immer vollständig so umsetzen, wie es in Scrum eigentlich vorgesehen ist. Es fanden zwar vergleichbare Termine statt, aber es war selten der Fall, dass eine Retrospektive im Scrum-Sinne zur Selbstreflektion des Entwicklerteams genutzt wurde und einen eigenständigen Termin zu dem Sprint Review darstellte. Meist war das Meeting zum Ende eines Sprints eine Mischform dieser beiden Scrum-Bestandteile. Dem grundsätzlichen Scrum-Ablauf hat dies in unserem Szenario jedoch keinen Abbruch getan, sodass Scrum für uns dennoch sehr gut funktioniert hat.

### **User-Story**

Die Aufgaben wurden als User Stories verfasst (aus Sicht des Benutzers) und anschließend in Tasks (kleinere Aufgaben) aufgebrochen. Diese wurden auf Karteikarten geschrieben und dem Scrum-Board hinzugefügt.

## 2.3. Technologien

Die bei unserem Projektseminar eingesetzten Technologien wurden zum größten Teil auf Grund von persönlicher Erfahrungen eines oder mehrerer Teammitglieder ausgewählt. Die Entscheidung, unser Programm hauptsächlich mit Java zu schreiben, beruht allein auf der Tatsache, dass alle Teammitglieder ausreichend viel Erfahrung mit dieser Programmiersprache hatten. Die Entscheidung für Java als Programmiersprache führte bei den Recherchen bezüglich der Kommunikation mit der Twitter-API zu der Java-Bibliothek Twitter4J [8]. Diese kapselt die Kommunikation mit Twitter komplett und liefert die Ergebnisse einer Anfrage an Twitter als einfaches Java-Objekt.

Die Entscheidung für eine bestimmte Datenbank war ebenfalls von Vorwissen im Team geprägt. MySQL ist die Datenbank, mit der alle Mitglieder bereits erste Erfahrungen gemacht hatten. Eine tatsächliche Evaluierung der zur Verfügung stehenden Datenbanken wurde nicht durchgeführt. Eine Auswahl einer Datenbank, die auf unsere Anforderungen besser spezialisiert ist, hätte allerdings unter Umständen aufgetretene Performanceprobleme besser lösbar gemacht.

Die Entscheidung, die Ausgabe unserer Analysen in einer Webseite darzustellen, begründet sich zum einen auf dem expliziten Wunsch des Teams, dies in unser Projekt mit aufzunehmen. Zum anderen erschien dem Team eine Benutzeroberfläche für eine Java-Applikation nicht zeitgemäß. Daher wurde entschieden, für das Frontend JavaScript in Verbindung mit jQuery [37] zu verwenden. Die Graphen in unserer Anzeige wurden mit dem Framework Highcharts erstellt [17]. Die Entscheidung für Highcharts beruhte in erster Linie wieder auf vorher vorhandene Erfahrungen einiger Teammitglieder. Dieses Framework ermöglicht, es einfache Graphen schnell und unkompliziert zu erstellen. Für die komplexeren Anzeigen, wie zum Beispiel die Tag-Cloud, wurde nach einer Recherche das Framework Data-Driven Documents (D3) verwendet [13]. Diese umfangreiche Visualisierungsbibliothek löst das Problem einer Visualisierung von Daten allerdings mit einer anderen Herangehensweise als Highcharts. Hierbei werden dem Entwickler alle Werkzeuge an die Hand gegeben, die benötigt werden, um eine beliebige Vektorgrafik zu erstellen und zu animieren.

## 2.4. Code-Organisation

Wie im vorherigen Kapitel beschrieben verlässt sich unser Projekt an vielen Stellen auf externe Bibliotheken. Um einen reibungslosen Entwicklungsablauf innerhalb des Teams zu erreichen, muss eine Methode bereitgestellt werden, welche die Verwaltung dieser externen Bibliotheken übernimmt. Diese Aufgabe wird für Java-Projekte von dem Programm Maven der Apache Foundation übernommen.

Bei diesem Programm handelt es sich um ein plattformunabhängiges Build-Management-Tool für Java-Projekte, das die Entwickler beim Anlegen, Kompilieren, Testen und Verteilen des Projekts unterstützt. Ein Großteil dieser Schritte soll automatisiert erledigt werden. Die Konfiguration all dieser Funktionen wird in einer XML Datei (`pom.xml`) festgehalten. An dieser Stelle lassen sich auch alle Abhängigkeiten zu externen Bibliotheken eintragen. Diese werden bei dem Kompilier-Befehl aus einem zentralen Repository in die Entwicklungsumgebung geladen, sowie alle Build-Path-Parameter entsprechend für das Projekt gesetzt. Dies erlaubt es, externe Bibliotheken in sehr einfacher Art und Weise in ein Projekt einzubinden, ohne einen komplizierten Importierungsprozess für den einzelnen Entwickler zu erfordern.

In dieser XML-Datei können ebenfalls die Optionen für die Tests des Projekts gesetzt werden. Über ein Plugin für verschiedene Entwicklungsumgebungen (Eclipse, IntelliJ) können die Konfigurationsdateien für diese Programme sehr einfach erstellt werden. Über Plugins der Entwicklungsumgebungen ist es allerdings auch möglich, Maven-Projekte in die Programme zu importieren.

In den ersten Wochen unseres Projektseminars wurden die beiden Komponenten Daemon und REST-Service als zwei separate Maven-Projekte angesehen. Erst als die Regressionsmodelle eine Verbindung zwischen den beiden notwendig gemacht haben, musste die Konfiguration umgestellt werden. Ein erster Versuch sah vor, das Daemon-Projekt lokal in das Maven-Repository zu installieren und es dann als direkte Abhängigkeit in das REST-Service-Projekt mit aufzunehmen. Diese Herangehensweise erwies sich aber als nicht praktikabel, da beim Kompilieren des REST-Service-Projekts der Code für die Daemon-Klassen nur dann neu aus dem Repository geladen wurde, wenn es sich um eine neue Version des Projekts handelte. Eine neue Versionsnummer für das Projekt musste allerdings vor jedem Kompilieren per Hand gesetzt werden. Aus diesem Grund wurde dieser Ansatz verworfen. Eine Lösung für dieses Problem konnte mit dem Modul-Plugin für Maven erreicht werden. Hierbei werden mehrere Maven-Projekte als einzelne Module

unter einem übergeordneten Projekt zusammengefasst. Der essentielle Punkt ist hierbei, dass ein Unterprojekt von einem anderen abhängig sein darf. Diese Konfiguration erlaubte es, die beiden Projekte nacheinander zu kompilieren und den REST-Service abhängig vom Daemon zu machen. Somit sind die Klassen des Daemon-Projekts auch im REST-Service nutzbar. Dies bedeutet ebenfalls, dass für einen Kompilierungsvorgang des gesamten Projekts sowohl die Tests im Daemon- als auch im REST-Service-Modul durchlaufen werden müssen. Um diese Tests unabhängig vom Entwicklungssystem bzw. unabhängig von dem gerade verwendeten Computer durchführen zu können, mussten einige Vorkehrungen getroffen werden.

## 2.5. Tests

Wie vorher erwähnt, ist eine Komponente der Softwareentwicklung, die über Maven automatisiert abläuft, das Testen. Bei dem Kompilierungsvorgang führt Maven ebenfalls alle Tests aus, die sich in einem vorher definierten Unterordner des Projekts befinden. Maven arbeitet hier nahtlos mit JUnit zusammen, um alle Tests in der entsprechenden Konfiguration durchzuführen.

Für den REST-Service waren vor allem die Integrationstests von essentieller Wichtigkeit. Hier wurden die verschiedenen Anfragen getestet, welche die Ergebnisse an das Frontend weiterreichen sollten. Um eine unabhängige Testumgebung zu erreichen, wurde hier eine explizite Testdatenbank eingerichtet, die vor allen Tests einmal aufgebaut wird. Hierzu wurde eine Datenbank per Hand bearbeitet und in eine Datei exportiert, die alle SQL-Anweisungen beinhaltet, um die Datenbank neu zu erstellen. Diese Datei konnte als Ressource in das Projekt eingebunden werden und ebenfalls zur Versionskontrolle hinzugefügt werden. Diese Dump-Datei musste noch für die lokale Datenbank angepasst werden. Der Name der Datenbank, der sich in der Dump-Datei befindet musste durch den in der `database.properties`-Datei ersetzt werden. Diese Vorkehrungen ermöglichten es, in den darauffolgenden Tests von einer sehr spezifischen Datenbank ausgehen zu können und somit komfortabel Tests zu erwarteten und unerwarteten Eingaben durchzuführen. Neben den Integrationstests wurden auch viele Methoden, die Datenbankabfragen weiterverarbeiten, mit Unit-Tests getestet. Auch hier wurden Randbedingungen, wie zum Beispiel leere Arrays, mit überprüft. Weitere Informationen zu den Besonderheiten des REST-Service sind im Infrastruktur-Kapitel bzw. in den Kapiteln der einzelnen Module zu finden.

Die Tests im Daemon, die eine externe Datenbank voraussetzen, wurden ebenfalls mit einer extra Datenbank realisiert. Es wurde erneut eine Datenbank per Hand erstellt und ein Dump erzeugt, der in die Versionierung mit aufgenommen wurde. Bei den Integrationstests wurde hauptsächlich getestet, ob von Twitter abgefragte Informationen korrekt in die Datenbank übertragen wurden. Da ein Teil der Sentimentanalyse auch im Daemon stattfindet, müssen die Unit-Tests für diese ebenfalls im Daemon Projekt abgehandelt werden. Eine weitere Komponente des Daemons, die ausgiebig getestet wurde, ist das Multithreading. Mehr Details zu der Funktionsweise des Multithreading und den Testfällen sind im Hauptkapitel des Daemons zu finden.

## 3. | Architektur

Aus der zuvor beschriebenen Projektidee ergeben sich drei Aufgabenbereiche von TMetrics: die Beschaffung und Filterung der Daten von Twitter, ihre Analyse durch Verfahren innerhalb von TMetrics (beispielsweise das Clustering oder die Analyse des Meinungsbildes) und die abschließende Darstellung der Resultate für den Benutzer in einer Browser-Anwendung (im Folgenden auch als Frontend bezeichnet). Die daraus resultierende Grundstruktur ist in Abbildung 3.1 sehen.

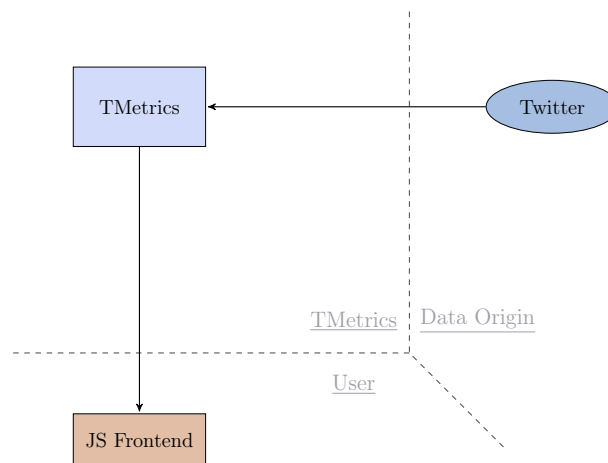


Abbildung 3.1.: Grundstruktur

### 3.1. Beschaffung der Daten

Recherchen zu Projektbeginn haben gezeigt, dass die grundsätzliche Möglichkeit zum Zugriff auf Tweets, Benutzerinformationen und sonstige Daten über die Twitter-API besteht. Diese Möglichkeit ist allerdings auch mit einer Reihe von Einschränkungen verbunden. Diese ermöglichen nur den Zugriff auf Tweets der letzten Tage und begrenzen außerdem die Anzahl der Tweets, welche innerhalb eines spezifischen Zeitraums zurückgeliefert



werden können. Das macht es notwendig, dass TMetrics die benötigten Daten in regelmäßigen Abständen von Twitter abrufen und in einer eigenen Datenbank speichert. Dies geschieht durch einen kontinuierlich laufenden Prozess, welcher von uns als Daemon bezeichnet wird. Da die Gesamtheit aller auf Twitter verfügbaren Daten unsere verfügbaren Kapazitäten überschreiten würde, haben wir uns dafür entschieden, auf diese Weise nur jene Daten zu sammeln, welche mit bereits eingegebenen Suchbegriffen in Verbindung stehen. Für diese Suchbegriffe war allerdings der Anspruch, sämtliche verfügbaren Daten zu sammeln, selbst wenn dafür zunächst noch keine Darstellung im Frontend vorgesehen war. Aus diesem Grund orientiert sich das Datenbankschema an den von Twitter zur Verfügung stehenden Daten. Um diese Entscheidungen zu realisieren, war eine Auswahl von Technologien erforderlich.

Bei der Wahl der Datenbank bestand Einigkeit über den Einsatz eines relationalen Datenbanksystems. Außerdem sollte ein System gewählt werden, mit dem alle Projektmitglieder Erfahrung hatten, da die Datenbank als zentraler Bestandteil der Projektarchitektur erkannt wurde, mit der viele Programmbestandteile interagieren müssen. Aus diesen Gründen fiel die Wahl auf MySQL, das sämtlichen Projektmitgliedern bekannt war. Des Weiteren gab es unter uns keinen Datenbankexperten, der eine besser geeignete Datenbank hätte vorschlagen können. Da das Datenbanksystem vor Projektbeginn nicht als wichtige Technologie erkannt wurde, bestand keine Zeit mehr für Recherchen zu Alternativen, da die dadurch entstandene Verzögerung wegen der zentralen Rolle der Datenbank den gesamten Projektfortschritt verzögert hätte. Da der Daemon in Java implementiert wurde und auf die Datenbank zugreifen sollte, war der Einsatz einer geeigneten Bibliothek als Schnittstelle notwendig. Aufgrund bestehender Erfahrungen fiel die Wahl dabei auf JDBC [4].

Neben einer Schnittstelle zur Datenbank war ebenfalls eine Schnittstelle zu Twitter notwendig, um die zu speichernden Daten überhaupt erst zu erhalten. Die bereits erwähnte API funktioniert dabei nach dem REST-Prinzip (siehe Abschnitt 4.3.1). Das bedeutet, dass der Zugriff auf bestimmte Arten von Daten mit einer bestimmten URL (Ressource) verbunden ist. So stellen beispielsweise die Tweets eine Ressource von Twitter dar, auf die dann mittels eines HTML-GET-Requests auf die entsprechende URL in Verbindung mit geeigneten Parametern (z. B. ein in dem Tweet vorkommender Suchbegriff oder Hashtag) zugegriffen werden kann. Die Antwort wird dann im JSON-Format zurückgeliefert. Allerdings ist diese API nicht öffentlich zugänglich, sondern erfordert einen Twitter-Account. Über diesen kann dann über das OAuth-Authentifizierungsprotokoll eine Reihe von To-

kens erstellt werden, welche bei der REST-Anfrage übergeben werden müssen, um eine Antwort zu erhalten. Um diese Schwierigkeit zu umgehen, aber auch um die Kommunikation mit der REST-API allgemein sicherer und einfacher zu gestalten, wurde auch für diese Schnittstelle nach einer Java-Bibliothek gesucht. Dabei fiel die Wahl auf Twitter4J. In dieser Bibliothek ist es möglich, durch einmalige Angabe der benötigten Tokens eine Verbindung mit Twitter herzustellen. Über diese Verbindung können dann REST-Anfragen gestellt werden, wobei dedizierte Methoden die sichere Übergabe der benötigten Parameter ermöglichen. Sämtliche Anfragen sind dann damit automatisch authentifiziert. Zu Beginn wurde zu Testzwecken zunächst ein Twitter-Account angelegt. Im Verlauf des Projekts waren allerdings aufgrund von Beschränkungen des Zugriffsvolumens weitere Accounts notwendig. Ein genauerer Überblick über diese Beschränkungen allgemein sowie die daraus resultierende Funktionsweise des Daemon wird im weiteren Verlauf dieser Arbeit im Abschnitt der Daemon-Komponente (siehe Abschnitt 4.1) geliefert.

## 3.2. Darstellung der Daten

Wie bereits erwähnt, sollen die gesammelten Daten auf Anfrage des Benutzers im Frontend (unter Umständen nach Abschluss zusätzlicher Analysen) dargestellt werden. Das Design des Frontends wird ausführlich im Abschnitt der entsprechenden Komponente beschrieben (siehe Kapitel 5.4). Die grundsätzliche Idee war jedoch, sich an der Gestalt von Suchmaschinen zu orientieren. Das Element der zusätzlichen Analyse und Aufbereitung der vorhandenen Daten legt außerdem einen Vergleich zu Seiten wie Wolfram Alpha nahe. Das bedeutet, dass der Benutzer zu Beginn einen Suchbegriff eingibt, der in den zu untersuchenden Tweets vorkommen soll. Damit der Daemon nach bereits angefragten Begriffen suchen kann, werden diese in der Datenbank in eine Tabelle eingetragen. Hat der Daemon zu dem Begriff bereits Daten gesammelt, können diese zur Darstellung zurückgeliefert werden. Dabei gibt es eine Reihe unterschiedlicher Gesichtspunkte, nach denen die vorhandenen Daten dargestellt werden können, beispielsweise das aktuelle Meinungsbild, die Aktivität zu dem Suchbegriff im zeitlichen Verlauf oder das Clustering. Die Ergebnisse zu jedem dieser Gesichtspunkte werden in einer entsprechenden Box, der sogenannten Ansicht, dargestellt. Dabei benötigt jede Ansicht eine entsprechende Auswahl der Daten aus der Datenbank sowie unter Umständen die Ergebnisse von deren Analyse. Damit ist klar, dass eine Schnittstelle notwendig ist, welche es ermöglicht, von Client-Seite Informationen in die Server-Datenbank einzutragen (vor allem zum Hinzufügen neuer Suchbegriffe) und

außerdem gemäß bestimmten Anfragen Daten aus der Server-Datenbank an den Client zu senden. Die Entscheidung fiel daher darauf, aufgrund von bereits bestehenden Erfahrungen von Projektmitgliedern sowie in Anlehnung an die bekannte Twitter-API einen REST-Service zu erstellen. Dieser wurde wie im Abschnitt der REST-Komponente (siehe Kapitel 4.3) näher beschrieben mithilfe der JAX-RS-Bibliothek Jersey als Tomcat-Servlet implementiert. Diese ermöglicht es, Ressourcen mit Java-Methoden zu verknüpfen, welche bei Aufruf der mit der Ressource verbundenen URL ausgeführt werden. Dabei wurde jede Ansicht mit einer Ressource verknüpft, welche die für die Ansicht benötigten Daten durch Aufruf einer GET-Anfrage zurückliefert. Die damit verknüpfte Methode beinhaltet dabei immer einen Zugriff auf die Datenbank über JDBC mittels einer geeigneten SQL-Anfrage. Informationen, welche die Anfrage einschränken, werden dabei als Parameter an die Anfrage angehängt und entsprechend in den SQL-Befehl eingefügt. Das Einfügen von Daten in die Datenbank geschieht analog über POST-Anfragen. Bei manchen Anfragen ist außerdem noch eine zusätzliche Analyse oder Aufbereitung der Daten notwendig, was im folgenden Abschnitt beschrieben wird. Diese finden nach Zugriff auf die Datenbank, jedoch vor Senden der Antwort an den Client durch das Jersey-Servlet statt. Als Format für die Antwort des REST-Services haben wir JSON gewählt, da es einen einfachen Zugriff durch JavaScript ermöglicht.

Die Notwendigkeiten zur Beschaffung und Darstellung der Daten liefert ein grundlegendes Architekturmodell von TMetrics bestehend aus Daemon und REST-Service, welches in Abbildung 3.2 dargestellt ist.

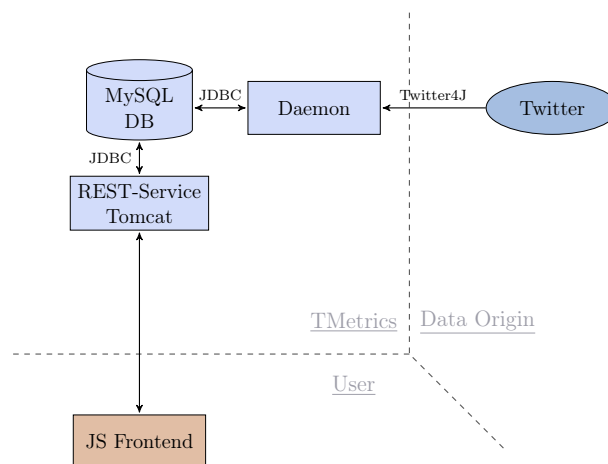


Abbildung 3.2.: Grundlegendes Architekturmodell

### 3.3. Analyse der Daten

Aus den beiden Hauptanwendungsbereichen Beschaffung der Daten von Twitter und Ausgabe der Daten an das Frontend ergibt sich die Projektstruktur von TMetrics: einerseits existiert ein REST-Service als Tomcat-Servlet auf dem Server, welcher auf Benutzeranfragen reagiert, und andererseits gibt es den Daemon, welcher kontinuierlich als gewöhnliche Java-Anwendung auf dem Server läuft. Dieser ursprünglichen Vorstellung nach bildet die MySQL-Datenbank die einzige Schnittstelle zwischen diesen beiden Bestandteilen. Damit waren zu Projektbeginn zwei grundsätzlich unabhängige Programme vorgesehen, welche an unterschiedlichen Stellen des Servers ausgeführt werden sollten. Aus diesem Grund wurden zwei separate Java-Projekte erstellt, welche im Folgenden Daemon und REST-Service genannt werden. Allerdings übersteigt der ursprüngliche Funktionsumfang von TMetrics die bisher vorgestellten Aufgabenbereiche dieser Bestandteile, welche sich jeweils nur auf das Auslesen und Speichern der Daten von Twitter und der Ausgabe dieser Daten an das Frontend beschränken. Hinzu kommt nämlich noch die Zielsetzung, dass die gesammelten Daten außerdem auf verschiedene Weisen analysiert werden sollen. Dies wirft die Frage auf, in welchem der Bestandteile diese Analyse stattfinden soll. Eine Durchführung im Daemon bedeutet dabei, dass die Analyse bereits bei Erhalt sämtlicher Daten durchgeführt wird und ihre Ergebnisse in der Datenbank persistent gehalten werden müssen. Im Gegensatz dazu wird eine Analyse im REST-Service nur durchgeführt, wenn eine entsprechende Anfrage des Benutzers gestellt wird. Das Speichern der dabei bestimmten Ergebnisse in der Datenbank ist möglich, aber nicht zwingend notwendig. Dabei muss in Betracht gezogen werden, wie häufig die Ergebnisse benötigt werden und wie schnell sie durch vom Daemon neu hinzugefügte Daten veralten können. Da das Sentiment eines Tweets nur von seinem Text abhängt, ist die Sentiment-Analyse nicht von neu hinzukommenden Tweets abhängig, muss daher nicht aktualisiert werden und kann daher direkt beim Hinzufügen des Tweets durch den Daemon durchgeführt werden (es ist möglich, dass das Sentiment aktualisiert werden muss, falls sich das zu Grunde liegende Modell ändert, was jedoch vergleichsweise selten vorkommt. Details dazu im Abschnitt der Sentiment-Analyse, Kapitel 5.1). Des Weiteren sind zur Erstellung des aktuellen Meinungsbildes eines Suchbegriffs ohnehin die Sentiment-Werte sämtlicher Tweets mit diesem Suchbegriff notwendig. Die Cluster-Analyse sowie die Bestimmung von Peaks und zugehörigen News in der Aktivitätskurve sind hingegen von sämtlichen Tweets in der Datenbank abhängig. Daher müssten die Ergebnisse dieser Verfahren ständig aktualisiert werden, so-

bald neue Tweets eines Suchbegriffs der Datenbank hinzugefügt werden. Da dies weitaus häufiger vorkommt als eine Anfrage an den Begriff, finden diese beiden Analysemethoden im REST-Service statt. Die Ergebnisse werden auf der Serverseite nach ihrem Abschicken an den Client nicht weiter gespeichert.

Einen Sonderfall bildet hierbei die Sentiment-Analyse. Nachdem im Projektverlauf mehr Gewicht auf den Aspekt des explorativen Arbeitens auf den zurückgelieferten Daten gelegt wurde, ist es nun auch wünschenswert, dass im Frontend angezeigt werden kann, welche Einflussfaktoren das Sentiment eines Tweets bestimmen und aus welchen Trainingsdaten diese Einflussfaktoren hervorgehen. Diese zusätzlichen Informationen gehen aus dem Sentiment-Modell hervor, welches wie zuvor beschrieben Teil des Daemon-Projekts ist. Zur Anzeige im Frontend müssen sie nun allerdings auch im REST-Service verfügbar sein. Eine Erweiterung der Datenbank um diese Einflussfaktoren wurde aufgrund des erhöhten Speicherbedarfs sowie der notwendigen Änderungen an der Datenbank mit Hinblick auf den Projektstand nicht vorgenommen. Stattdessen wurde entschieden, dass für Tweets, für die die Einflussfaktoren abgerufen werden, das Sentiment noch einmal erneut im REST-Service bestimmt wird. Der dadurch entstehende Zusatzaufwand ist hinnehmbar, da davon auszugehen ist, dass dies nur für einen Bruchteil der Tweets in der Datenbank durchgeführt wird. Damit ist allerdings der Einsatz der Klassen des Sentiment-Modells im REST-Service notwendig, sodass keine strikte Trennung beider Projekte mehr gegeben ist. Die Integration der Analysemethoden in das Architekturmodell ist in Abbildung 3.3 zu sehen.

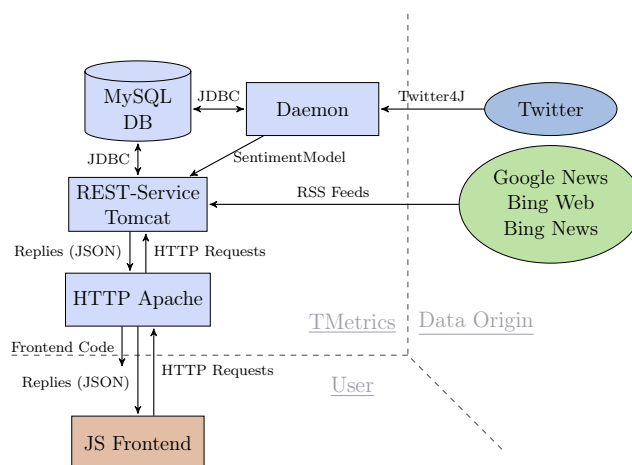


Abbildung 3.3.: Architekturmodell mit Analysemethoden

## 4. | Infrastruktur

### 4.1. Daemon

Wie schon erwähnt, ist eines unserer Ziele, Tweets von Twitter zu bestimmten Suchbegriffen, bei uns *search terms* genannt, zu sammeln und weiterzuverarbeiten. Da wir aufgrund einiger Restriktionen durch Twitter nicht direkt alle Tweets zu einem Thema bekommen können, ist es notwendig sie zwischenzuspeichern. Der Service, der die Tweets von Twitter abrufen und in der lokalen Datenbank speichert, wurde in Anlehnung an Unix-Dienste Daemon genannt. Im folgenden Kapitel 4.1.1 wird auf die Twitter-API sowie die damit verbundenen Restriktionen eingegangen. Anschließend wird in Abschnitt 4.1.2 erklärt, wie die Tweets von Twitter abrufen werden und insbesondere welche Suchstrategie dabei angewendet wird, um möglichst alle Tweets zu einem Thema zu finden. Es folgt in Abschnitt 4.1.3 eine Beschreibung der Erweiterung der Suche zur parallelen Suche, um den Durchsatz der gefundenen Tweets zu erhöhen. In diesem Rahmen wird auch das mit der Suche eingeführte Multi-Threading erläutert. Als letzte Verbesserung des Daemon folgt in Abschnitt 4.1.4 die Erläuterung wie möglichst zeitnah nach Eingabe eines neuen Suchbegriffs Ergebnisse erhalten werden. Als letztes folgt im Abschnitt 4.1.5 eine Diskussion über weitere mögliche Erweiterungen der Fähigkeiten des Daemon, die nicht mehr umgesetzt werden konnten. Das Sentiment wird zwar im Daemon berechnet, wird aber nicht hier beschrieben, sondern in Kapitel 5.1, welches speziell nur das Thema Sentiment behandelt.

#### 4.1.1. Twitter-API

Die Twitter-API umfasst alle Möglichkeiten über die zum Teil unterschiedlichen offiziellen Schnittstellen mit Twitter zu interagieren. Sie bildet alle Funktionen ab, die notwendig sind, um einen Twitter-Client zu bauen. Um die API verwenden zu können, benötigt man sowohl ein sogenanntes Profil, d. h. einen Account bei Twitter, als auch einen API-Key, den man selbst erzeugen kann. Twitter unterscheidet zwischen dem sogenannten *consumer key*,

der an das Profil gebunden ist, und dem *application key*, der an die Anwendung gebunden ist. Dabei kann ein Nutzer verschiedene Apps nutzen, die alle einen eigenen *application key* nutzen, sein *consumer key* ändert sich aber nicht. Genauso kann ein Entwickler mehrere Apps entwickeln, die alle einen unterschiedlichen *application key* besitzen, sein *consumer key* bleibt aber gleich. Ein Nutzer muss bei einer App allerdings nicht selbst diese Keys eintragen, vielmehr fordert die App, die den *application key* im Code enthält, den User auf, die App bei Twitter zu autorisieren. Die Details lassen sich in der Twitter-Dokumentation, Abschnitt „Access Token“ [27] entnehmen. Das Authentifizierungssystem für Apps von Twitter basiert dabei auf OAuth, auch hier finden sich wieder die Details in der Twitter-Dokumentation [26]. Unser Daemon greift im Gegensatz dazu etwas anders auf Twitter zu. So haben wir für jedes genutzte Profil eine eigene App bei Twitter registriert und die jeweiligen *consumer* und *application Keys* direkt verwendet. Diese Art des Zugriffs ist eigentlich nur für Entwickler vorgesehen.

## Beschränkungen der Twitter-API

Grundsätzlich bietet Twitter zwei unterschiedliche APIs: die REST-API und die Streaming-API. Bei der REST-API müssen explizite Anfragen gestellt werden, um Daten zu erhalten, während bei der Streaming-API zu gewissen Themen oder Usern ständig die aktuellen Daten übermittelt werden, solange man eine Verbindung offen hält. Die einzige für uns relevante Funktionalität ist die Suche nach den durch den Nutzer vorgegeben Begriffen, deswegen beschränken wir uns auf die Funktionalität zum Suchen nach Tweets. Das ist grundsätzlich mit beiden APIs möglich, die Streaming-API würde es uns aber nur erlauben, Tweets zu dem gesuchten Begriff zu erhalten, die nach unserer Anfrage erstellt wurden. Das folgt aus der zweiten Grafik in der Dokumentation der Twitter Streaming-API [30]. Es ist mit der Streaming-API also nicht möglich Tweets zu finden, die zeitlich vor Beginn unserer Anfrage erstellt wurden. Die REST-API hingegen erlaubt es uns auch nach Tweets zu suchen, die schon vor Beginn unserer Anfrage erstellt wurden. Allerdings gibt es auch hier Beschränkungen. So garantiert Twitter nicht, dass man Tweets findet, die älter als ca. 6-9 Tage sind, obwohl es durchaus ältere Tweets geben kann. Die Dokumentation zur Nutzung der Suche [33] sagt dazu

The Search API is not a complete index of all Tweets, but instead an index of recent Tweets. At the moment that index includes between 6-9 days of Tweets.

Ebenfalls hier wird auch die Beschränkung erläutert, dass Twitter keine Vollständigkeit der Ergebnisse garantiert, sondern nur davon spricht, die möglichst relevanten Ergebnisse zu liefern. Weitere Beschränkungen sind:

- maximal 100 Tweets pro Anfrage und
- maximal 180 Anfragen pro 15-Minuten-Intervall pro Account (siehe *REST API v1.1 Limits per window by resource* in der Twitter Dokumentation [29]).

Eine Einführung in diese *rate limits* genannten Beschränkungen findet sich im Artikel *REST API Limiting in v1.1* ([28]) der offiziellen Dokumentation. Für uns ist außerdem noch relevant, dass Twitter es nicht ermöglicht die Suche durch kleinere Intervalle als Tage einzuschränken. Dafür ist es möglich die Suche durch Angabe einer maximalen bzw. minimalen Tweet-ID einzuschränken. Diese sowie weitere mögliche Parameter bei der Suche sind im *GET search/tweets* Artikel [25] der Twitter-Dokumentation zu finden. Ebenfalls hier werden die *result types* eingeführt. Die *result types* geben dabei an, von welcher Art das Ergebnis ist. Mögliche Werte sind:

- **popular** - Enthält nur die populärsten Tweets
- **recent** - Enthält die aktuellsten Tweets
- **mixed** - Enthält beide Arten von Tweets

Dabei liefert der *result type recent* nicht nur aktuelle Tweets, sondern diese auch in einer chronologischen Sortierung. Wir konnten in eigenen Tests keine nicht-chronologische Abfolge von Tweets feststellen. Außerdem scheint **recent** es uns zu erlauben alle Tweets zu erhalten, die Twitter uns zur Verfügung stellt. Wir konnten jedenfalls keine fehlenden Tweets feststellen, als wir mit den verschiedenen Strategien experimentiert haben. Unsere Tests erfolgten dabei unter der Annahme, dass die mit **recent** erhaltenen Tweets streng chronologisch sind. Leider waren wir nicht in der Lage, eine genaue Aussage zur Chronologie und Vollständigkeit der verschiedenen *result types* in der Twitter-Dokumentation zu finden.

#### 4.1.2. Suchstrategie

Um Tweets von Twitter systematisch abzufragen, wird eine intelligente Suchstrategie benötigt. Die Strategie muss sich dabei den durch Twitter auferlegten Einschränkungen



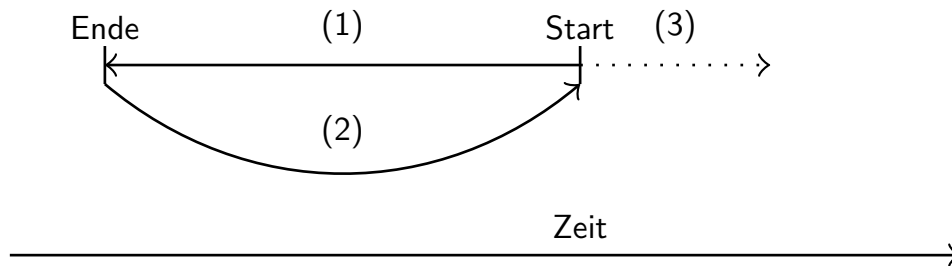


Abbildung 4.1.: Der erste Ansatz. Zunächst wird vom Start aus rückwärts gesucht (1). Anschließend wird wieder zum ursprünglichen Start gesprungen (2). Von dort aus wird dann vorwärts nach neuen Tweets gesucht (3).

anpassen. Beispielsweise ist es mit der Search-API, die Teil der bereits erwähnten REST-API ist und für die Suche nach Tweets verwendet wird, nicht möglich, nach Tweets in Zeiträumen von Stunden oder Minuten zu suchen. Stattdessen bietet Twitter mit der Search-API die Suche nach Tweets in Zeitintervallen von Tagen oder die Suche bis zu einer Tweet-ID an. Letzteres bedeutet, dass eine Anfrage zu einem Suchbegriff an Twitter mit einer Tweet-ID gesendet werden kann, in der der Benutzer die 100 letzten Tweets zu diesem Suchbegriff mit einer ID kleiner als der in der Anfrage verwendeten ID erhält.

Im Daemon gab es zwei Ansätze, wie die Suche durchzuführen sein sollte. Der erste Ansatz sah vor, dass in der ersten Phase ab dem Zeitpunkt  $t$ , an dem ein neuer Suchbegriff in die Datenbank geschrieben wurde, bei Twitter nach Tweets gesucht wird, die vor  $t$  veröffentlicht wurden. Werden anschließend keine neuen Tweets mehr gefunden, deren Erstelldatum vor  $t$  liegt, so wird nun in einer zweiten Phase ab  $t$  nach neuen Tweets gesucht. Abbildung 4.1 verdeutlicht diesen Ansatz.

Der alternative Ansatz unterscheidet sich in der zweiten Phase vom ersten Ansatz. Anstatt ab Zeitpunkt  $t$  in Richtung Gegenwart zu suchen, wird der Zeitpunkt  $t$  auf „jetzt“ neu gesetzt. Anschließend wird wieder nach Tweets gesucht, die vor dem neuen  $t$  veröffentlicht worden sind, bis anschließend keine neuen Tweets mehr gefunden werden. Dann wird  $t$  erneut gesetzt und die Suche beginnt abermals. Dieser Ansatz besteht also aus vielen, aneinandergereihten Einzelsuchen. In Abbildung 4.2 wird der zweite Ansatz skizziert.

Wir haben uns für den zweiten Ansatz entschieden, da sich dieser gut mit den Einschränkungen durch Twitter verträgt und keine Phasen unterscheiden muss, wie es im ersten Ansatz der Fall ist. Aus diesem Grund wurde auch auf die Verwendung der Streaming-API

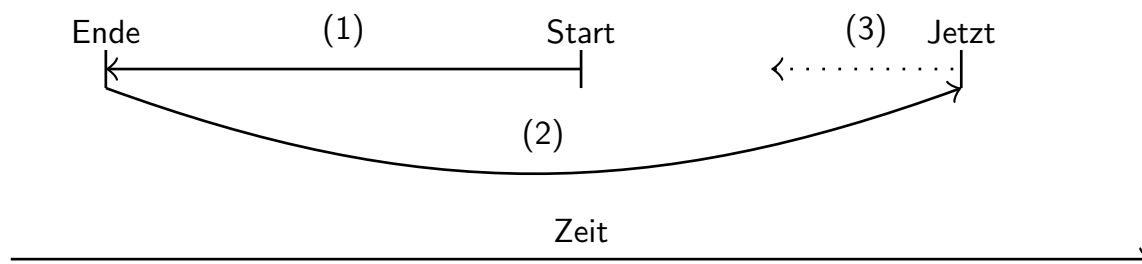


Abbildung 4.2.: Der alternative Ansatz. Zunächst wird vom Start aus rückwärts gesucht (1). Anschließend wird zum jetzigen Zeitpunkt gesprungen (2). Von dort aus wird wieder rückwärts gesucht (3).

verzichtet, da der Daemon stets rückwärts sucht und nicht über gerade neu veröffentlichte Tweets sofort informiert werden muss.

### Details der Suchstrategie

Auch wenn die im Daemon implementierte Suchstrategie keine unterschiedlichen Phasen besitzt, so gibt es aber doch unterschiedliche Zustände, in denen sich die Suchlogik befinden kann. Wie bereits erwähnt, besteht die Suche aus aneinandergereihten Einzelsuchen. Diese Einzelsuchen haben wir *Iterationen* genannt. Eine Iteration kann sich in zwei Zuständen befinden. Hat sie noch nicht begonnen, dann wurde noch nicht nach neuen Tweets, die vor dem Zeitpunkt  $t$  veröffentlicht worden sind, gesucht. In der Iteration kann jedoch auch schon nach neuen Tweets gesucht worden sein, deren Erstelldatum vor  $t$  liegt. In diesem Fall ist die Iteration *aktiv*.

Bevor auf genaue Details der Suchlogik eingegangen werden kann, muss zunächst der Begriff der *Intervalllänge* erläutert werden. Die Intervalllänge ist ein Zeitraum, der mit einem Suchbegriff assoziiert ist. Dieser Zeitraum bestimmt, wann erneut nach diesem Suchbegriff gesucht werden soll. Denn nach Begriffen, die wenig Aktivität erfahren, es also über einen längeren Zeitraum nur wenige neue Tweets gibt, sollte weniger häufig gesucht werden als nach Begriffen, die eine starke Aktivität besitzen, es also permanent viele neue Tweets gibt. Dies ist insbesondere deswegen sinnvoll, weil ein für die Suche verwendetes Twitter-Profil nur beschränkt viele Anfragen innerhalb von 15 Minuten zur

Verfügung hat. Dieser Sachverhalt wird im Abschnitt über das *scheduling* weiter vertieft werden.

Zu Beginn der Suche ist die Iteration nicht aktiv. Der Suchzeiger, der angibt, wo wir uns auf dem Zeitstrahl der Iteration befinden, steht also auf dem Zeitpunkt  $t$ . In diesem Fall muss die Intervalllänge des zu bearbeitenden Suchbegriffs betrachtet werden. Ist der Zeitraum, der zwischen der letzten Suche und  $t$  liegt, kleiner als die Intervalllänge, so darf nicht nach dem Suchbegriff gesucht werden. Wir setzen in diesem Fall  $t$  neu auf den jetzigen Zeitpunkt, sodass bei einem erneuten Prüfen, ob gesucht werden soll, die Möglichkeit besteht, dass die Intervalllänge nun überschritten worden ist. In dem Fall wird dann einmal nach Tweets gesucht, deren Erstelldatum vor  $t$  liegt. Anschließend werden für die Suche nur noch Tweets-IDs verwendet, da Twitter keine Suche nach exakten Zeitpunkten zulässt. Dazu werden Anfragen an Twitter gestellt, die neben dem Suchbegriff auch die Tweet-ID  $id$  beinhalten. Twitter liefert dann für den Suchbegriff bis zu 100 Tweets, deren IDs kleiner als  $id$  sind. Gleichzeitig sind diese Tweets chronologisch sortiert und liegen zeitlich direkt vor dem Erstellzeitpunkt des Tweets von  $id$ . Aus diesen den erhaltenen Tweets wird anschließend der älteste ausgewählt, dessen ID als neuer Suchanker dient. So handelt sich der Suchzeiger anhand der Tweet-IDs zeitlich rückwärts entlang, bis keine neuen Tweets mehr gefunden werden. In dem Fall wird  $t$  neu gesetzt und die aktuelle Iteration beendet. Insbesondere wird die Intervalllänge nicht beachtet, wenn die Iteration aktiv ist.

Ein kleines Beispiel verdeutlicht diesen Ablauf: Wir befinden uns am Anfang der Iteration, sie ist also noch nicht aktiv. In  $t$  steht der momentane Zeitpunkt und die Intervalllänge wird überschritten. Also darf gesucht werden. Dazu wird für einen Suchbegriff eine Anfrage an Twitter gestellt, die 100 jüngsten Tweets vor dem jetzigen Zeitpunkt zu liefern. Twitter ermöglicht zwar keine Suche zu bestimmten Zeitpunkten, bietet jedoch die Funktionalität, die 100 jüngsten Tweets vor dem aktuellen Zeitpunkt anzufordern. Alle Tweets werden zwischengespeichert, jedoch noch nicht in der Datenbank abgespeichert (siehe dazu Abschnitt den Abschnitt über die Architektur des Daemons). Aus diesen Tweets wählen wir den ältesten aus und merken uns seine Tweet-ID. In einer nächsten Anfrage liefert uns Twitter die 100 jüngsten Tweets vor dem Tweet mit der gemerkten ID. Auch hier wählen wir wieder den ältesten aus, den wir für eine neue Anfrage nutzen können. Dies wird so lange fortgeführt, bis keine neuen Tweets mehr gefunden werden. Das kann zwei Ursachen haben: Zum einen kann es sein, dass die Iteration einen viel zu langen Zeitraum überstreckt (also mehr als 6-9 Tage). Dann liefert Twitter grundsätzlich keine

Tweets mehr aus. Also muss die Iteration beendet werden. Zum anderen kann es aber auch sein, dass das komplette Intervall abgearbeitet wurde und wir Tweets finden, deren Erstelldatum bereits vor dem Startzeitpunkt der vorhergegangenen Iteration liegen. Diese Tweets liegen schon bereits alle in der Datenbank, weswegen die Iteration beendet werden kann.

## Berechnung der Intervalllänge

Die Intervalllänge gibt an, wann zu einem Suchbegriff gesucht werden soll. Ist noch nicht genug Zeit seit dem letzten Iterations-Startzeitpunkt vergangen, so wird noch nicht nach dem Suchbegriff gesucht. Denn ansonsten würde sehr häufig nach Suchbegriffen gesucht werden, zu denen es keine bis nur sehr wenige, neue Tweets gibt. So würden unnötig viele Suchanfragen gestellt und die Twitter-Profile deswegen nicht optimal ausgenutzt werden. Es erscheint daher sinnvoll, nur dann nach Suchbegriffen zu suchen, wenn davon ausgegangen werden kann, mit einer einzigen Suchanfrage alle neuen Tweets zu finden. Da eine Anfrage höchstens 100 Tweets liefert, muss die Anzahl der mit einer Anfrage aufzufindenden Tweets also auf 100 gesetzt sein. Nach einem Suchbegriff, zu dem in der Stunde im Schnitt nur zwei neue Tweets veröffentlicht werden, sollte also ungefähr alle 50 Stunden gesucht werden. Falls der Benutzer jedoch zeitiger Ergebnisse sehen möchte, kann er die Intervalllänge über die Benutzer-Priorisierung für den Suchbegriff reduzieren. Analog können nicht so wichtige Suchbegriffe über eine negative Priorisierung eine längere Intervalllänge erhalten.

Um festzustellen, wie häufig neue Tweets zu einem Suchbegriff vorkommen, wird während der gesamten Suche der jüngste und der älteste gefundene Tweet festgehalten (nicht der jüngste und älteste einer Anfrage). Über die über den gesamten Zeitraum zwischen ältesten und jüngsten Tweet gefundenen, neuen Tweets lassen sich die Tweets pro Minute (*TPM*) ermitteln. Anhand der *TPM*-Zahl lässt sich der Zeitraum bestimmen, über welchem erwartungsgemäß weitere 100 neue Tweets vorhanden sein werden. Da die Anzahl der neuen Tweets aber immer etwas schwankt und nicht konstant gleich bleibt, gibt es noch einen Korrekturwert, der mit der Intervalllänge multipliziert wird. Der Wert wurde auf 0,9 festgelegt. Veranschaulicht werden dann nur 90 der 100 neuen Tweets gefunden, da nur 90% des Zeitraums abgedeckt werden. Dieser Faktor soll sicherstellen, dass leichte Abweichungen nach oben nicht eine weitere Anfrage provozieren. Stattdessen wird der Puffer von noch 10 Tweets verwendet. Zukünftig wird die Intervalllänge anschließend gekürzt, da nicht so viel Zeit wie durch die Intervalllänge vorgegeben ist, vergehen muss.

Falls es hingegen eine Abweichung nach unten gibt, so wird die Intervalllänge verlängert werden, da mehr Zeit verstreichen muss, bis 90 Tweets gefunden werden.

Insgesamt ergibt sich die Intervalllänge  $I$  gemäß folgender Formel:

$$I = p \cdot \frac{1}{TPM} \cdot t \cdot 100,$$

wobei  $p \in \{0,5; 0,75; 1; 1,5; 2\}$  der Benutzer-Priorität von positiv (0,5 und 0,75) über neutral (1) bis negativ (1,5 und 2) entspricht,  $TPM$  die Tweets pro Minute sind und  $t$  dem *throttle factor* von 0,9 entspricht. Die 100 entspricht den 100 angepeilten, neuen Tweets, die es über den Zeitraum der Intervalllänge zu finden gilt.

Für den Fall, dass kein oder nur 1 Tweet gefunden wurde, wird die momentane Intervalllänge mit einem „Außenseiter“-Faktor multipliziert, die aktuell auf 3 gesetzt ist. Das Eintreten dieses Falls ist unerwartet, da die Intervalllänge ja auf 90 Tweets ausgelegt war. Aus diesem Grund wird sie verlängert, da der Suchbegriff offenbar ein unerwartetes Ereignis durchlebt haben muss, was die Intervalllänge vorher hat falsche Annahmen machen lassen. Die Intervalllänge kann insgesamt eine Länge von sechs Tagen erreichen, nicht jedoch mehr. Dieser Wert wurde gewählt, da Twitter Tweets nur bis zu sechs Tagen in die Vergangenheit garantiert. Dies stellt zwar nicht sicher, dass alle Tweets in dem Zeitraum der letzten Tage gefunden werden (weil vielleicht eine plötzliche, unerwartete Aktivität mit dem Suchbegriff verbunden ist), aber ein noch längeres Warten wäre sinnlos, da dann tatsächlich Tweets verloren gehen würden. Da die Zeit für das Erneuern des *rate limits* durch Twitter auf 15 Minuten festgelegt ist, sollte die Intervalllänge mindestens eine Länge von 15 Minuten besitzen. Aus diesem Grund wird eine Intervalllänge, die laut der Formel kürzer als 15 Minuten lang ist, automatisch auf 15 Minuten festgelegt.

## Scheduling

Die Anzahl der Suchanfragen innerhalb von 15 Minuten ist durch Twitter beschränkt, weswegen die Anzahl der Suchanfragen zu einer effizient zu verwaltenen Ressource wird. Aus diesem Grund wurde im Daemon ein intelligentes Verhalten zur effizienten Verwaltung der Suchbegriffe eingefügt, um möglichst nur dann eine Suche zu einem Suchbegriff zu starten, wenn dies auch sinnvoll erscheint.

Die Grundidee ist es nun, dass der Daemon immer nur dann eine Anfrage zu einem Suchbegriff startet, wenn er die 100 neuen Tweets seit dem letzten Abrufen mit einer Anfrage sammeln kann. Dies ist eine signifikante Verbesserung der Auslastung von Suchbegriff-

fen im Vergleich zum ursprünglichen Verhalten, bei dem der Daemon zu jedem aktiven Suchbegriff immer mindestens eine Anfrage bei Verwendung eines frischen Twitter-Profiles.

Um ein intelligentes Scheduling-Verhalten der Suchgriffe zu gewährleisten, werden Suchbegriffe in zwei Klassen unterteilt: *short search terms* und *long search terms* (auch einfach nur *short terms* und *long terms*).

Unter *short terms* verstehen wir solche Suchbegriffe, die eine Intervalllänge *länger* als die minimale, durch Twitter vorgegebene *rate limit* Resetzeit besitzen, was zur Zeit 15 Minuten sind. Das bedeutet, dass zu diesem Suchbegriff idealerweise innerhalb von einer Anfrage alle neuen Tweets gefunden und gespeichert werden können, denn die Intervalllänge ist ja gerade so kalkuliert worden, dass mit einer Anfrage 100 neue Tweets gefunden werden sollten. Da davon ausgegangen wird, dass mit einer Anfrage alle neuen Tweets gefunden werden, werden diese auch nur einmal abgefragt.

Für den Fall, dass mehr als 100 Tweets vorhanden, aber nicht abgefragt worden sind, wird die Intervalllänge entsprechend korrigiert, sodass die nächste Abfrage einer neuen Iteration bereits nach kürzerer Zeit stattfindet. Da allerdings noch weitere, neue Tweets vorhanden sind, werden bei weiteren Suchen unabhängig von der Intervalllänge ab der zuletzt gefundenen Tweet-ID weitere Tweets ermittelt, da sich der Suchbegriff in einer aktiven Iteration befindet. Alle Suchvorgänge werden von einzelnen Threads, die von uns als **Minions** bezeichnet und in Abschnitt 4.1.3 ausführlich erläutert werden, durchgeführt.

Gibt es beispielsweise einen *short term*, nach dem gesucht werden soll, so wird dieser *short term* einem neuen gestarteten **Minion** zugewiesen. Dieser sucht dann ein einziges Mal nach Tweets zu diesem Suchbegriff. Unabhängig davon, wie viele Tweets gefunden wurden, wird die Intervalllänge zu diesem Suchbegriff angepasst. Nachdem sich der **Minion** beendet hat, weil er entweder sein *rate limit* aufgebraucht hat oder alle Iterationen zu seinen zugewiesenen Suchbegriffen beendet hat, wird der *short term* erneut einem weiteren **Minion** zugeordnet. Dieser **Minion** sucht auf jeden Fall noch einmal nach Tweets zu diesem *short term*, da der *short term* sich momentan in einer aktiven Iteration befindet. Jetzt gibt es allerdings zwei mögliche Situationen. Zum einen kann es vorkommen, dass der **Minion** keine neuen Tweets mehr zu dem Suchbegriff findet. In dem Fall ist die Iteration beendet. Findet er hingegen noch neue Tweets, so darf die Iteration noch nicht beendet werden. Dann werden weitere **Minions** zu dem *short term* nach neuen Tweets suchen, bis die Iteration beendet werden kann.

Im oben genannten Szenario kann es durchaus passieren, dass ein *short terms* während einer Iteration zu einem *long term* wird, nämlich genau dann, wenn die Intervalllänge auf

15 Minuten korrigiert wird. Dies ist auch wünschenswert, da der *short term* offenbar eine Veränderung in der Aktivität durchläuft und somit nicht mehr als *short term* bezeichnet werden sollte. Alle *short terms* werden vor den *long terms* abgearbeitet, da ansonsten die *long terms* alle Anfragen des Twitter-Profiles aufbrauchen könnten und somit nicht mehr nach den *short terms* gesucht werden könnte.

Neben den zuvor genannten *short terms* gibt es noch *long terms*. Unter *long terms* verstehen wir solche Suchbegriffe, die eine Intervalllänge von genau 15 Minuten besitzen. Das bedeutet, dass angenommen wird, dass zu diesem Suchbegriff innerhalb von 15 Minuten mindestens 100 neue Tweets auftreten. Aus Erfahrung kann jedoch davon ausgegangen werden, dass es deutlich mehr als 100 neue Tweets sind. Aus diesem Grund werden für die nach Abarbeiten der *short terms* verbliebenen *long terms* auch alle restlichen Anfragen reserviert.

Ein *long term* wird, anders als ein *short term*, nicht nur einmal von einem **Minion** abgearbeitet, sondern so lange bis alle Anfragen aufgebraucht sind oder die Iteration zu dem Suchbegriff beendet werden kann. So ist gewährleistet, dass *long terms* der großen Datenmenge an Tweets auch nachkommen. Die *long terms* werden gemäß einer *round robin* Strategie durchgegangen, sodass jeder Suchbegriff annähernd gleich viele Anfragen erhält.

Der Name *long term* rührt daher, dass nach diesen Suchbegriffe lange gesucht werden muss, während *short terms* nur eine kurze Suchzeit besitzen. Eine mögliche alternative Interpretation der Namen bezüglich der Dauer der Intervalllänge wurde nicht gewählt.

Die Aufgabe des Daemons ist es, permanent Daten zu allen aktiven Suchbegriffen zu sammeln. Dabei ist es durchaus sinnvoll, jüngere Suchbegriffe ein wenig zu priorisieren, da zu diesen Suchbegriffen möglichst schnell ein repräsentativer Datenbestand aufgebaut werden soll. Gleichzeitig besitzen ältere Suchbegriffe bereits einen relativ zum Suchbegriff ausreichend großen Datenbestand.

Aus diesem Grund werden sowohl die Liste der *short terms* als auch *long terms* zeitlich sortiert, sodass am Anfang der Listen der jüngste Suchbegriff steht. Am Beispiel der *long terms* werden wir sehen, weswegen diese Sortierung eine leichte Priorisierung bewirkt.

Wir stellen uns das folgende Szenario vor. Es sind fünf *long terms* vorhanden, wovon ein Suchbegriff jung und die restlichen Suchbegriffe alt sind. Zusätzlich stehen nur noch zwölf Anfragen zur Verfügung. Die Liste wird nun zeitlich sortiert, sodass der junge Suchbegriff ganz am Anfang steht. Er bekommt die erste Anfrage. Anschließend sind die älteren Suchbegriffe an der Reihe. Jetzt hat jeder Suchbegriff einmal eine Anfrage stellen dürfen

und es sind noch sieben Anfragen verfügbar. Bei einem weiteren Durchlauf erhält ebenfalls wieder jeder Suchbegriff eine Anfrage. Jetzt stehen jedoch nur noch zwei Anfragen zur Verfügung, das heißt nicht jeder Suchbegriff kann eine Anfrage stellen. Es ist daher sinnvoll, die verbliebenen Anfragen auf die Suchbegriffe zu verteilen, die es noch am Nötigsten haben, ihren Datenbestand zu erweitern. Dies ist bei jüngeren Suchbegriffen der Fall. Aus diesem Grund werden die zwei übrig gebliebenen Anfragen auf die ersten beiden Suchbegriffe verteilt. Darunter ist auch der junge Suchbegriff. Jetzt hat dieser Suchbegriff eine Anfrage mehr als fast alle anderen Suchbegriffe erhalten und wurde somit gemäß seines Alters priorisiert.

Die Priorisierung durch die zeitliche Sortierung ist jedoch nur eine Art, wie jüngere Suchbegriffe bevorzugt werden. Eine viel stärkere Priorisierung ergibt sich durch so genannte virtuelle Kopien von zu priorisierenden Suchbegriffen. Gemäß der Einteilung eines Suchbegriffs in jung (weniger als einen Tag alt), etwas älter (weniger als eine Woche, aber mindestens einen Tag alt) und alt (mehr als eine Woche alt) werden zu den Suchbegriffen unterschiedlich viele virtuelle Kopien erstellt und, ebenfalls zeitlich sortiert, an das Ende der Liste der von einem *Minion* abzuarbeitenden Suchbegriffe angehängt. Diese virtuellen Kopien werden wie ganz normale *long terms* behandelt, sodass dem zugrundeliegenden Suchbegriff insgesamt mehr Anfragen zukommen. Für *short terms* gibt es diese Priorisierung nicht, da ja davon ausgegangen wird, dass mit einer Anfrage bereits alle neuen Tweets gefunden wurden.

Ein Beispiel soll diesen Sachverhalt verdeutlichen. Wir haben wieder fünf Suchbegriffe gegeben, wobei einer davon jung, ein anderer etwas älter und die restlichen Suchbegriffe alt sind. Wenn wir annehmen, dass junge Suchbegriffe insgesamt pro *round robin* Runde  $a$  Anfragen, etwas ältere Suchbegriffe insgesamt  $b$  Anfragen und alte Suchbegriffe insgesamt  $c$  Anfragen erhalten sollen, wobei  $a \geq b \geq c$  gilt, sähe die Liste nach Priorisierung durch virtuelle Kopien wie in Abbildung 4.3 dargestellt aus.

Wie der Abbildung zu entnehmen ist, besitzen junge Suchbegriffe insgesamt  $(a - c)$  Anfragen mehr als alte und  $(a - b)$  Anfragen mehr als etwas ältere Suchbegriffe. Zudem haben etwas ältere Suchbegriffe  $(b - c)$  Anfragen mehr zur Verfügung als alte Suchbegriffe. Durch diese Priorisierung anhand von virtuellen Kopien wird der Datenbestand für jüngere Begriffe schneller aufgefüllt als bei älteren.

Momentan ist die Aufteilung für virtuelle Kopien wie folgt:

- Der Suchbegriff ist weniger als einen Tag alt: 2 weitere virtuelle Kopien



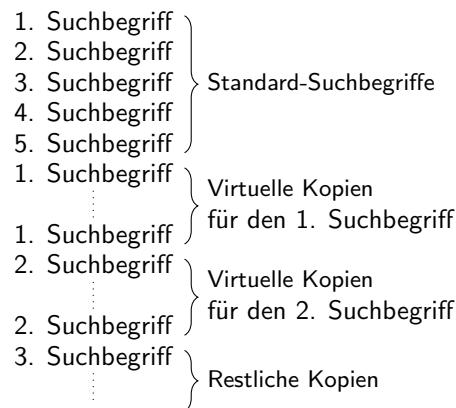


Abbildung 4.3.: Die Aufteilung der virtuellen Kopien zu den einzelnen Suchbegriffen. Diese Liste wird anschließend an einen *Minion* zur Abarbeitung übergeben.

- Der Suchbegriff ist weniger als eine Woche, aber mindestens einen Tag alt: 1 weitere virtuelle Kopie
- Der Suchbegriff ist älter als eine Woche: keine weiteren virtuellen Kopien

Die Suchstrategie des Daemons zu einem Suchbegriff wird also beeinflusst durch

- die Intervalllänge für des Suchbegriffes,
- die durch den Benutzer vorgegebene Priorität zu dem Suchbegriff,
- das Aktivitätsverhalten des Suchbegriffs, das den Suchbegriff als *short* oder *long term* klassifiziert und
- das Alter des Suchbegriffs, was zu einer unterschiedlichen Anzahl an virtuellen Kopien und somit zu einer unterschiedlichen Anzahl von Anfragen zum Suchbegriff führt.

### 4.1.3. Parallele Suche

#### Motivation

Nachdem der Daemon in der Lage ist mit einer gewissen Intelligenz Tweets zu suchen, könnte man annehmen der Daemon sei fertig. Für Untersuchungen im kleineren Umfang ist das prinzipiell auch korrekt, uns war die Menge der gefundenen Tweets aber nicht hoch genug. Wie sich aus den Beschränkungen der Twitter-API, die in Abschnitt 4.1.1

beschrieben wurden, ergibt, kann der Daemon mit einem Profil maximal 18.000 Tweets in 15 Minuten sammeln. Das ist vergleichsweise wenig, besonders wenn die Anzahl der Suchbegriffe steigt und auch sehr aktive Suchbegriffe, das heißt Suchbegriffe zu denen es aufgrund gesteigerten Interesses sehr viele Tweets gibt, dabei sind. Ein Beispiel für einen sehr aktiven Suchbegriff war *walker*. Mit diesem *search term* konnte man sehr viele Tweets zum Tod von Schauspieler Paul Walker, am 30. November 2013 finden, insgesamt ca. 11 Millionen innerhalb weniger Tage. Diese Zahlen berichtete zumindest Topsy<sup>1</sup>, sie sind aber leider nicht mehr abrufbar, allerdings findet sich im November/Dezember 2013 auch bei Google Trends [2] ein entsprechender Ausschlag im Interesse. Die Anzahl Tweets war zu groß, als dass unser Daemon sie alle hätte finden können. Das liegt daran, dass der Daemon, wie in Abschnitt Twitter-API 4.1.1 beschrieben, maximal 18.000 Tweets in 15 Minuten abrufen kann und wir über die Suche maximal 6-9 Tage zurückliegende Tweets finden können. Da wir noch mehr als diesen einen Suchbegriff bearbeiten, ist es in dem zur Verfügung stehenden Zeitfenster von 6-9 Tagen also nicht möglich alle Tweets zu diesem Thema zu erfassen. Deswegen wurde als nächstes Ziel entschieden, den Durchsatz zu erhöhen. Der für uns nächstliegendste Ansatz war der Einsatz mehrerer Profile, immerhin erlaubt jedes weitere Profil den Abruf von bis zu 18.000 weiteren Tweets. Um mehrere Profile optimal auszunutzen, müssen sie parallel genutzt werden. Das erlaubt sowohl die parallele Suche nach verschiedenen Suchbegriffen, als auch nacheinander mit verschiedenen Profilen nach einem Suchbegriff zu suchen. Da die Suche selbst auch einige Zeit in Anspruch nimmt, kann dies nicht beliebig gesteigert werden, erlaubt aber doch eine gewisse Skalierung. Gerade auch im Bezug auf eine steigende Zahl von Suchbegriffen erlaubt dies eine höhere Anzahl von gefundenen Tweets.

## Architektur

Das Parallelisieren erfolgt in einer klassischen Master-Worker-Architektur, bei der ein Master Aufgaben an mehrere Worker verteilt und ihre Ergebnisse sammelt. Dies erfordert eine Überarbeitung unserer bestehenden Architektur. Unser Master heißt auch **Master**, während wir unsere Worker **Minions** genannt haben. Außerdem soll der Daemon jetzt dauerhaft laufen und nicht wie vorher nur alle 15 Minuten durch einen *cronjob* gestartet werden. Dabei ist der **Master** für die Verwaltung der Profile und die Aufteilung der

---

<sup>1</sup>[www.topsy.com](http://www.topsy.com)

Suchbegriffe auf die **Minions** zuständig, während die **Minions** selbst nur die Tweets von Twitter abrufen und speichern sollen.

### Architektur: Master

Die Suchbegriffe holt sich der **Master** aus der Datenbank, nachdem er sie in *short* und *long terms* eingeteilt hat (siehe Suchstrategie 4.1.2), erhält der erste **Minion** mit einem gültigem Profil, d. h. mit einem Profil das noch freie Suchanfragen hat, eine bestimmte Anzahl davon und beginnt mit der Suche. Wie der **Master** Suchbegriffe aus der Datenbank abrufen und an die **Minions** verteilt ist in Grafik 4.4 dargestellt. Dabei merkt der **Master** sich, welche Suchbegriffe er schon verteilt hat, und registriert auch, wenn ein **Minion** mit seiner Arbeit fertig ist, seine Suchbegriffe also wieder frei werden. Ursprünglich ging der **Master** davon aus, dass alle Profile gültig sind, wenn er gestartet wird. Falls dem nicht so war konnte das dazu führen, dass der **Master** in einer Art Endlos-Schleife lief, dies aber nur temporär. In der Schleife hat der **Master** immer wieder einen **Minion** erzeugt. Dieser hat sich aber sofort wieder beendet, weil er keine Anfragen mehr frei hatte. Da der **Master** die Profile der Reihe nach durchgeht, hängt er so lange in der Schleife bis das Profil wieder gültig wird. Um diese Problematik zu beheben, wurde die Abfrage der noch freien Anfragen von den **Minions** in den **Master** verlagert. Dadurch weiß der **Master** immer, ob ein Profil gültig ist. Wenn der **Master** keine gültigen Profile oder keine aktiven Suchbegriffe mehr hat, schläft er für einen gewissen, einstellbaren Zeitraum, bevor er erneut prüft, ob es aktive Suchbegriffe und gültige Profile gibt.

### Architektur: Minions

Die Hauptaufgabe der **Minions** ist das Suchen und Speichern von Tweets. Große Teile dieser Aufgaben konnten direkt aus dem Code der vorherigen **single-threaded** Anwendung übernommen werden. Die Suchbegriffe, die jeder **Minion** bearbeitet, erhält er dabei direkt vom **Master**, sodass sich die **Minions** nicht darum kümmern müssen, welche Begriffe aktiv und noch nicht von anderen **Minions** belegt sind. Der **Master** erhält die Suchbegriffe dabei über den **Transactor**, der Schnittstelle zur Datenbank, direkt aus der Datenbank. Sobald ein **Minion** alle seiner Anfragen aufgebraucht hat, hat er sie über den **Transactor** in der Datenbank gespeichert. Es musste sichergestellt werden, dass nicht mehrere **Minions**, oder ein **Minion** und der **Master** gleichzeitig oder in der falschen Reihenfolge auf den gleichen Suchbegriff zugreifen. Schlimmstenfalls könnte es sonst passieren, dass zwei **Minions** in

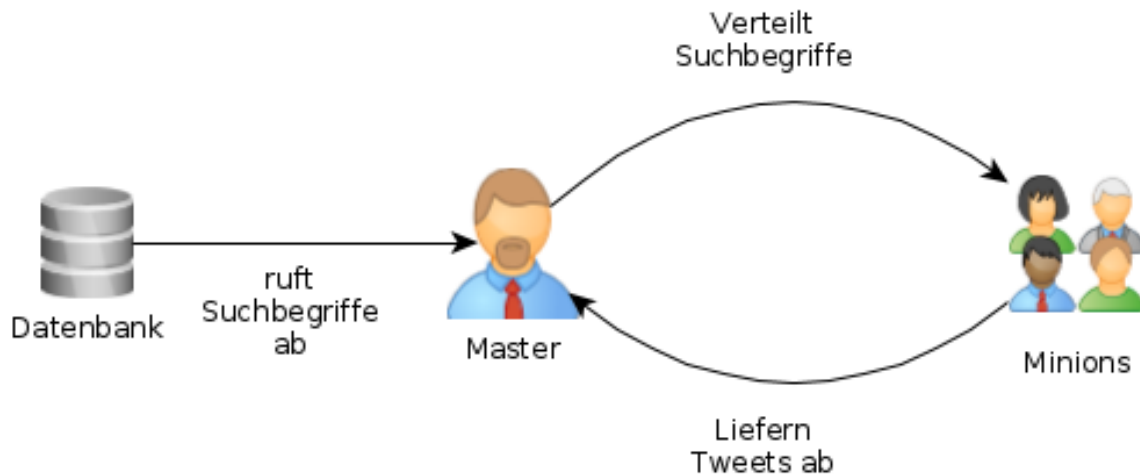


Abbildung 4.4.: Der **Master** holt Suchbegriffe aus der Datenbank und verteilt sie an die **Minions**.

der falschen Reihenfolge versuchen, denselben Suchbegriff zu updaten und das Programm dabei abstürzt. Dies würde zu inkonsistenten Werten bei dem Suchbegriff und damit verlorenen Tweets, d. h. Tweets die wir zwar finden könnten, aber nicht erfasst haben, führen. Auch würden so Anfragen verschwendet, wenn der **Master** einen neuen **Minion** zu einem Suchbegriff gestartet hat, der mit genau den gleichen Parametern schon vom vorherigen **Minion** bearbeitet wurde, diese Werte aber noch nicht in der Datenbank aktualisieren konnte. Ebenfalls stellte sich schnell heraus, dass die Datenbank-Performance erheblich darunter gelitten hat, wenn zwei **Minions** gleichzeitig ihre Tweets gespeichert haben.

### Architektur: Treasurer

Um das Problem zu lösen, entschieden wir uns dafür, das Speichern der Tweets und das Aktualisieren der Suchbegriffe an eine zentrale Stelle auszulagern. Das geschieht, indem die **Minions** die Tweets, die sie erhalten haben, zusammen mit den entsprechenden und aktualisierten Suchbegriffen beim **Master** abliefern. Der **Master** aktualisiert daraufhin sein lokales **SearchTerm**-Objekt und speichert die erhaltenen Tweets zwischen. Dabei stellen alle Tweets inklusive des dazugehörigen Suchbegriffs und seiner Parameter ein **Package** dar. Alle **Packages** werden in einem Beutel gesammelt und nach dem Datum des letzten Abrufs sortiert. Der Beutel ist dabei, ähnlich wie die **Treasury** ein virtuelles Konstrukt, in diesem Fall für einen Zwischenspeicher, um **Packages** zu sammeln. So stellen wir sicher, dass die **Packages** in der richtigen Reihenfolge in der Datenbank gespeichert werden. Falls

wir das nicht täten, könnten wir Tweets verlieren. Wenn der **Master** von der Datenbank neue Suchbegriffe erhält, vergleicht er sie mit seinen lokalen Objekten. Solange seine lokalen Objekte neuere Werte haben als die Werte, die die Datenbank liefert, werden die lokalen verwendet. Ausschlaggebendes Kriterium dafür, wie aktuell ein Suchbegriff ist, ist der Zeitpunkt der letzten Suche.

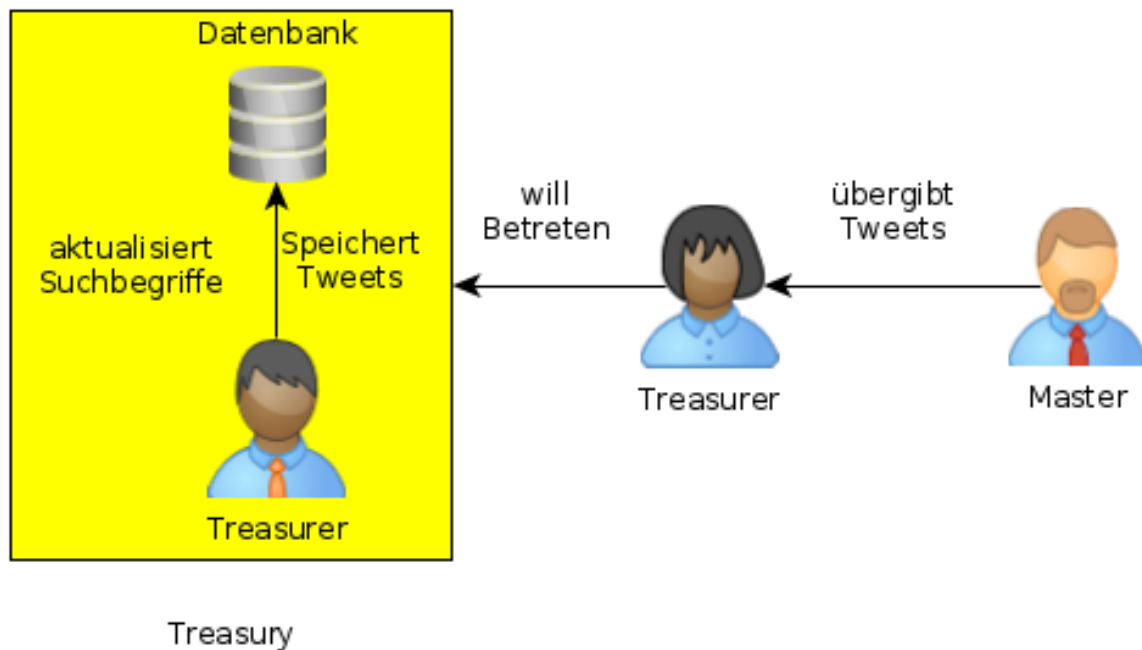


Abbildung 4.5.: Der **Master** übergibt die gesammelten Tweets einem **Treasurer**. Ein weiterer **Treasurer** speichert Tweets in die Datenbank.

Der **Master** soll nicht selbst die Tweets speichern, da er zum einen weiterhin neue Tweets der **Minions** entgegennehmen, zum anderen auch die Suchbegriffe verteilen muss. Um dies zu umgehen, wurde ein neuer **Worker** erschaffen, dessen einzige Aufgabe das Speichern von Tweets ist. Wir haben ihn **Treasurer** genannt, analog dazu gibt es auch eine **Treasury**, eine Schatzkammer. Die **Treasury** ist dabei ein virtuelles Konstrukt, das den Zugang zur Datenbank bildlich darstellt. Der **Master** wartet entweder, bis er mindestens eine gewisse Anzahl von Tweets in seinem Beutel gesammelt hat, oder bis eine einstellbare Zeit nach dem Ende des letzten **Treasurer** vergangen ist, bis er einen neuen **Treasurer** erzeugt. Es kann dabei durchaus mehrere **Treasurer** gleichzeitig geben, wenn sehr viele Tweets in kurzer Zeit gefunden werden, oder die alten **Treasurer** noch nicht mit ihrer Arbeit fertig sind, wenn ein neuer erzeugt wird. Der **Treasurer** erhält eine Kopie

des Beutels vom **Master** und dieser leert anschließend seinen Beutel. Nun versucht der **Treasurer** die **Treasury** zu betreten. Wie der Master die gesammelten Tweets an einen **Treasurer** übergibt, während ein weiterer **Treasurer** Tweets in der Datenbank speichert, ist in Grafik 4.5 dargestellt. Die **Treasury** ist dabei als Mutex realisiert und stellt somit auch eine Warteschlange für die **Treasurer** da. Versucht ein **Treasurer** die **Treasury** zu betreten, heißt das, dass er versucht das Mutex zu erlangen. Gelingt ihm das Betreten der **Treasury**, also das Erlangen des Mutex, speichert er alle Tweets aus seinem Beutel in der Datenbank, genau wie es der alte Daemon vorher getan hat. Außerdem aktualisiert er die **SearchTerms** in der Datenbank, allerdings erst, nachdem er die zu dem Suchbegriff gehörenden Tweets gespeichert hat. Falls der Daemon abstürzt oder beendet wird, bevor der Suchbegriff aktualisiert wurde, sind die Tweets trotzdem vorhanden und werden schlimmstenfalls erneut gefunden. Würde aber erst der **SearchTerm** aktualisiert und danach erst die Tweets gespeichert, wären alle Tweets verloren, die nicht mehr gespeichert wurden. Das liegt daran, dass der Daemon aufgrund seiner Suchstrategie davon ausgeht, dass die Tweets schon gespeichert sind. Anschließend informiert er gegebenenfalls andere wartende **Treasurer** über das Ende seiner Arbeit und beendet sich. Sollte die **Treasury** belegt sein, schläft ein **Treasurer** und wartet auf eine Benachrichtigung durch den die **Treasury** belegenden **Treasurer**. In der letzten Scrum-Iteration des Projektseminars wurde die Art, wie der **Treasurer** die Tweets in der Datenbank speichert, angepasst. Waren es zuerst jeweils einzelne Tweets, die gespeichert wurden, wurde das System auf *batch inserts* umgestellt. Dabei fügt der **Treasurer** erst eine bestimmte Anzahl von Tweets zu einem *batch* zusammen und speichert dann alle Tweets einer *batch* auf einmal. Dies brachte teilweise erhebliche Geschwindigkeitsvorteile, auch abhängig von der Größe der *batch*.

## Multi-Threading

Die parallele Suche an sich wurde mit einem klassischen Master-Worker Modell mit Multi-Threading umgesetzt. Der **Master** hat dabei hauptsächlich die verwaltende Tätigkeit, wie oben beschrieben. Generell ist sowohl der **Master** als auch jeder **Minion** und **Treasurer** ein eigener Thread. Ein Abstürzen eines einzelnen Threads sollte, solange es sich nicht um den **Master** handelt, nicht das ganze Programm abstürzen lassen. Es gab aber an einigen Stellen Schwierigkeiten mit dem Multi-Threading Ansatz. So müssen die einzelnen Worker mit dem **Master** interagieren und z. B. melden, wenn sie mit ihrer Arbeit fertig sind. Auch gibt es mehrere Stellen, an denen mehrere Threads um die gleiche Resource kämpfen, wo es Probleme mit Deadlocks gab. Ein klassisches Problem war dabei,

dass in der **Treasury** der Mutex nicht überall in der gleichen Reihenfolge erlangt wurde, dies konnte zu Deadlocks führen. Die Lösung war die Reihenfolge, in der die Mutexe erlangt werden, überall zu vereinheitlichen. Ein weiteres Problem konnte auftreten, wenn der **Master** keine freien Suchbegriffe hatte. Dabei hat er eine **if**-Abfrage abgebrochen, in dessen Rahmen ein Mutex erlangt wurde, ohne es frei zu geben. Da die **Minions** aber denselben Mutex benötigten, um ihre Tweets und Suchbegriffe abzugeben, kam es zu einem Deadlock. Das Finden dieses Fehlers war sehr aufwändig. Die Lösung, dass der **Master**, bevor er die Schleife abbricht, noch den Mutex freigibt, ist dagegen recht trivial. Das Multi-Threading erforderte auch die Einführung der schon oben erwähnten **Packages**. Alle Tweets, die ein **Minion** findet, sind mit dem Suchbegriff, zu dem sie gefunden wurden, sowie den zugehörigen Metadaten zum Zeitpunkt der Suche verknüpft. Wenn nun nacheinander mehrere **Minions** denselben Suchbegriff bearbeiten, gäbe es mehrere Tweets, die alle zum selben Suchbegriff gehören, aber mit unterschiedlichen Metadaten assoziiert sind. Es war also nicht möglich alle Tweets zu einem Suchbegriff im Beutel des **Master** zu sammeln. Stattdessen wurden **Packages** eingeführt. Jedes **Package** stellt dabei eine Sammlung der Tweets mit dem assoziierten Suchbegriff eines Suchdurchlaufs dar. Die einzelnen **Packages** werden dann nach dem Zeitpunkt der Suche chronologisch sortiert und auch in derselben Reihenfolge abgespeichert. So wird verhindert, dass im Falle eines Absturzes des Programms Tweets verloren gehen, wenn sie weder gespeichert sind, noch erneut gesucht werden.

#### 4.1.4. Zeitnahe Ergebnisse

Bislang musste der Benutzer im schlimmsten Fall bis zu 15 Minuten auf Ergebnisse warten, wenn er einen neuen Suchbegriff eingegeben hat. Denn hatten alle zur Suche verwendeten Profile kurz bevor der Benutzer die Suche zu einem neuen Suchbegriff startete ihr *rate limit* ausgeschöpft, so musste er warten, bis wieder ein Profil frei wurde. Dies konnte aber wie bereits erwähnt dann bis zu 15 Minuten dauern. Das war jedoch nicht besonders benutzerfreundlich und wurde daher als Problem angesehen, das es zu beheben galt.

#### Motivation

Es wäre daher besser, wenn der Benutzer einen neuen Suchbegriff eingibt und anschließend nach möglichst kurzer Zeit, maximal nach 20 bis 30 Sekunden, die ersten Ergebnisse seiner Suchanfrage angezeigt bekommt. Die Anzahl der Ergebnisse muss noch nicht sehr hoch

sein; vielmehr genügt es dem Benutzer, wenn er zumindest einen kleinen Datensatz als Grundlage zum Analysieren besitzt. So kann er sich bereits sehr zeitnah einen ersten Überblick über den Suchbegriff verschaffen. Zu einem späteren Zeitpunkt haben sich dann wesentlich mehr Daten zum Suchbegriff gefunden, mit denen der Benutzer dann intensiver arbeiten kann.

## Konzept

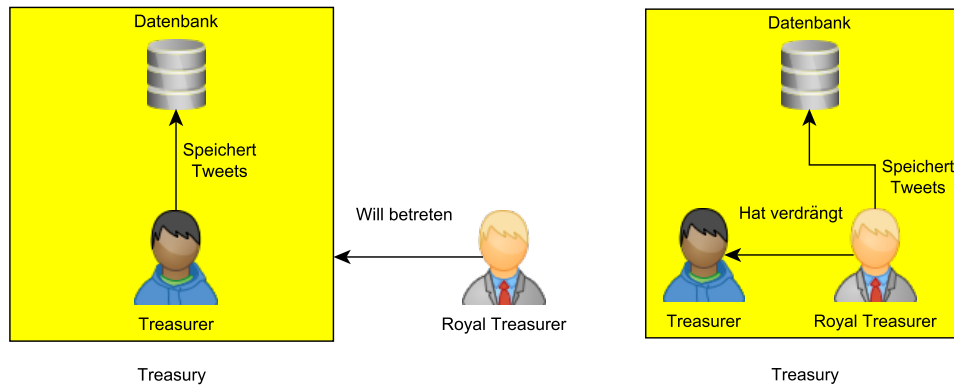
Es stellt sich nun die Frage, wie erreicht werden kann, dass bereits sehr zeitnah zur Suchanfrage die ersten Daten präsentiert werden können. Um das oben geschilderte Problem der aufgebrauchten Twitter-Profile zu umgehen, führen wir ein weiteres Profil ein, das nur genutzt wird, wenn ein neuer Suchbegriff in der Datenbank gefunden wird. So ist sichergestellt, dass es immer ein Profil gibt, das nach Tweets zum neuen Suchbegriff suchen kann. Des Weiteren darf dieses gesonderte Profil jedoch nicht sofort für einen neuen Suchbegriff das *rate limit* vollständig aufbrauchen, weswegen die Anzahl der Anfragen an Twitter durch den Daemon begrenzt wird. Wäre die Anzahl der Anfragen nicht begrenzt, so stünden für einen weiteren, neuen Suchbegriff keine Anfragen mehr zur Verfügung und der Benutzer müsste wieder 15 Minuten warten. Die maximal zulässige Anzahl an Anfragen für das gesonderte Profil lässt sich beliebig einstellen; ein Grenzwert von 10 hat sich jedoch als akzeptabel herausgestellt. In diesem Fall verbraucht das Profil also lediglich 10 der 180 Anfragen, weswegen noch 170 Anfragen zur Verfügung stehen. Es kann also innerhalb von 15 Minuten nach höchstens 18 neuen Suchbegriffen gesucht werden. Da es jedoch unwahrscheinlich ist, dass in diesem kurzen Zeitraum tatsächlich so viele neue Suchbegriffe hinzukommen, ist diese Limitierung als nicht-kritisch anzusehen.

Mit 10 Anfragen lassen sich bis zu 1000 Tweets zu einem neuen Suchbegriff finden, was eine akzeptable erste Datenbasis schafft. Möchte man mehr Tweets finden, so muss der Grenzwert der Anfragen an Twitter hochgestellt werden. In diesem Fall können jedoch nicht 18 neue Suchbegriffe innerhalb von 15 Minuten gefunden werden, sondern es sind weniger. Ein Verringern des Grenzwertes liefert zwar weniger Tweets, doch kann der Benutzer dann mehr als 18 neue Suchbegriffe innerhalb der 15 Minuten eingeben.

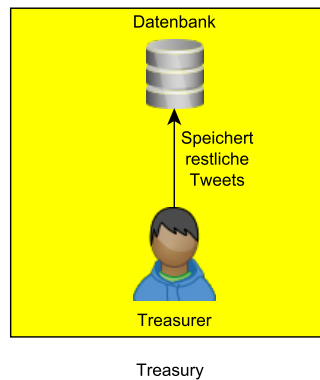
Es sollte jedoch bedacht werden, dass ein geringer Grenzwert mit wenigen Anfragen auch weniger Zeit für die Suche nach Tweets benötigt als ein höherer Grenzwert mit mehr Anfragen.

Das besondere Profil alleine hilft allerdings noch nicht dabei, dass der Benutzer auch zeitnah die gefundenen Tweets angezeigt bekommt. Bislang ist lediglich sichergestellt, dass





- (a) Ein Royal Treasurer will die Treasury betreten. (b) Der Royal Treasurer verdrängt den Treasurer.



- (c) Der verdrängte Treasurer setzt seine Arbeit fort.

der Daemon auf jeden Fall nach einem neuen Suchbegriff suchen kann. Bis die Ergebnisse allerdings dem Benutzer auch tatsächlich präsentiert werden können, kann jedoch ebenfalls einige Zeit vergehen, da die Tweets noch nicht in der Datenbank gespeichert wurden. Stattdessen werden sie zunächst im Beutel des **Masters** hinterlegt, welcher jedoch nicht zwingend umgehend durch einen **Treasurer** geleert werden muss. Da der **Treasurer** eine vorgegebene Zeit wartet, bis er den Beutel des **Masters** leert, oder er den Beutel leert, falls die festgelegte Zahl der Tweets im Beutel überschritten wird, kann es teilweise sehr lange dauern, bis die neuen Tweets in die Datenbank geschrieben wurden. Der Benutzer musste also weiterhin lange auf die ersten Ergebnisse warten.

Um das Problem zu umgehen, wird ein weiterer Beutel eingeführt, der ausschließlich für die zeitnahen Tweets genutzt wird. Wurde dieser Beutel befüllt, wird er – unabhängig davon, wann er zuletzt befüllt worden ist oder wie voll er ist – sofort von einem **Treasurer** geleert, damit die Tweets auch sofort in die Datenbank geschrieben werden können. Somit wurde die Wartezeit bis zur Leerung des Beutels eliminiert. Dies hilft allerdings immer noch nicht zwingend. Denn es kann durchaus vorkommen, dass bereits ein anderer **Treasurer** in der **Treasury** aktiv ist und unser wichtigerer **Treasurer** warten muss, was bedeutet, dass auch der Benutzer warten muss. Aus diesem Grund erhält der **Treasurer**, der den besonderen Beutel leert, eine höhere Priorität gegenüber anderen **Treasurern**. Um die Höhe der Priorität anzudeuten, wird dieser **Treasurer** deswegen **Royal Treasurer** genannt. Betritt dieser nun die **Treasury**, verdrängt er einen möglicherweise aktiven **Treasurer** und kann somit seine Tweets sofort abspeichern. Ist er mit dem Abspeichern fertig, so weckt er den verdrängten **Treasurer** wieder auf, sodass dieser seine Arbeit an dem Punkt fortsetzen kann, an dem er unterbrochen wurde. Die Abbildungen 4.6a, 4.6b und 4.6c illustrieren diese Situation.

Es kann vorkommen, dass es zeitweise mehrere **Royal Treasurer** gibt, nämlich genau dann, wenn mehrere Suchbegriffe zeitnah neu eingegeben werden. Tritt diese Situation ein, so werden die **Royal Treasurer** sequentiell in der Reihenfolge, in der sie die **Treasury** betreten, aktiv. Ein möglicherweise zuvor verdrängter **Treasurer** muss dann die gesamte Zeit lang warten, bis alle **Royal Treasurer** fertig sind.

## Implementierung

Das besondere Profil wird von einem besonderen **Minion** genutzt, der darauf achtet, dass der zuvor gesetzte Grenzwert von beispielsweise zehn Anfragen auch beachtet wird. Die anderen **Minions** suchen nämlich so lange zu Suchbegriffen, bis ihr *rate limit* aufgebraucht ist. Der besondere **Minion** wird von uns **Limited Minion** genannt, da er nur eine begrenzte (engl.: *limited*) Anzahl an Anfragen an Twitter stellt. Der **Limited Minion** teilt, nachdem er seine Suche abgeschlossen hat, dem **Master** über eine gesonderte Nachricht mit, dass er fertig ist. Aufgrund dieser gesonderten Nachricht weiß der **Master** anschließend, dass es sich um den **Limited Minion** handelt, weswegen der **Master** entsprechend ein sofortiges Leeren des besonderen Beutels durch den **Royal Treasurer** einleitet.

Dieser **Royal Treasurer**  $R$  betritt nun die **Treasury** und versucht das Mutex der **Treasury** zu erhalten. Es kann jedoch sein, dass bereits irgendein (Royal) **Treasurer**  $A$  aktiv in der **Treasury** arbeitet und  $R$  das Mutex nicht erhalten kann, weswegen er kurzzei-

tig warten muss. Ist  $A$  ein anderer **Royal Treasurer**, so muss  $R$  weiterhin warten, da er andere **Royal Treasurer** nicht verdrängen darf. Ist  $A$  hingegen ein normaler **Treasurer**, so darf  $R$   $A$  verdrängen. Dies tut  $R$  jedoch nur indirekt. Während nämlich  $A$  seine Tweets in kleineren Paketen abspeichert, schaut er immer wieder nach, ob in der **Treasury** ein **Royal Treasurer** wartet. Falls dies der Fall ist, gibt  $A$  das Mutex frei und legt sich schlafen, bis er wieder durch einen fertig gewordenen **Royal Treasurer** aufgeweckt wird. Hat  $A$  nun das Mutex freigegeben, blockiert  $R$  es sofort und beginnt anschließend, seine Tweets abzuspeichern. Ist  $R$  fertig, informiert er den schlafenden  $A$  darüber und weckt diesen auf. Anschließend gibt  $R$  das Mutex wieder frei, welches  $A$  dann sofort wieder sperrt. Sicherheitshalber prüft  $A$  anschließend sofort noch einmal, ob nicht schon wieder ein anderer **Royal Treasurer** wartet. Ist dies der Fall, legt sich  $A$  direkt wieder schlafen.

(**Royal**) **Treasurer** können das **Treasury**-Mutex nicht direkt verwenden. Vielmehr wird das Betreten oder Verlassen der **Treasury** und das damit einhergehende Sperren und Entsperren des Mutexes durch die **Treasury** verwaltet. So ist die Konsistenz innerhalb des Arbeitsflusses unter den einzelnen (**Royal**) **Treasurern** gewährleistet.

#### 4.1.5. Diskussion

Obwohl der Daemon gemäß der Kundenwünsche als fertig bezeichnet werden kann, hätte er dennoch um einige Aspekte erweitert werden können, sofern mehr Zeit zur Verfügung gestanden hätte.

##### Verbesserte parallele Suche

Bislang ist es nicht möglich, dass mehrere **Minions** zu einem Suchbegriff parallel suchen. Dies wäre jedoch wünschenswert, um beispielsweise Suchbegriffe, die temporär eine extreme Aktivität verzeichnen (beispielsweise aufgrund eines unvorhersehbaren Vorkommnis in Zusammenhang mit dem Suchbegriff) besser abzarbeiten. Besonders problematisch wird es, wenn zu diesen Suchbegriffen innerhalb von wenigen Stunden eine Anzahl von mehreren Millionen Tweets anfällt. Denn dann kann der Daemon in seiner bisherigen Form die große Datenlast nicht vollständig abarbeiten, da nach 6-9 Tagen die Tweets von Twitter nicht mehr bereitgestellt werden, der Daemon jedoch noch nicht alle Tweets der gesteigerten Aktivität gesammelt hat. So gehen unerwünschterweise Daten verloren.

Könnten jedoch stattdessen mehrere **Minions** parallel zu einem solchen Suchbegriff suchen, steigt die Bearbeitungsgeschwindigkeit der Tweets linear in der Anzahl der parallel

arbeitenden **Minions**. Eine Umsetzung dieser Idee stellt sich jedoch als schwierig dar, da Twitter es nicht ermöglicht, eine Suche auf Zeitintervalle geringer als einen Tag einzuschränken.

Dies ist jedoch nötig, da die riesige Datenmenge nicht nur über Tage gestreut ist, sondern eben bereits über Stunden, teilweise sogar über Minuten. Daher ist eine Beschränkung der Suche auf Stunden- oder Minutenintervalle wünschenswert. Da dies jedoch nicht möglich ist, muss als alternativer Weg die Suche über die IDs einzelner Tweets stattfinden. Hierbei wäre es die grundlegende Idee, den verstrichenen Zeitraum  $s$  zwischen dem ältesten ( $t_o$ ) und dem neusten Tweet ( $t_n$ ) einer einzelnen Suchanfrage zu betrachten. Anschließend berechnet man die Differenz  $d$  der beiden IDs der betrachteten Tweets  $t_o$  und  $t_n$  und geht nun davon aus, dass über einen Zeitraum  $s$  insgesamt  $d$  viele neue Tweets veröffentlicht werden. Nun wird dieser Zeitraum  $s$   $n$ -mal in die Vergangenheit zurück gegangen bis zu dem Zeitpunkt, ab dem man einen weiteren **Minion** suchen lassen möchte, wobei das  $n$  selbst gewählt wird. Um nun tatsächlich ab diesem Zeitpunkt zu suchen, wird die zuvor erhaltene Differenz  $d$  ebenfalls  $n$ -mal von der ID des Tweets  $t_n$  subtrahiert. Ab dieser ID kann dann ein neuer **Minion** suchen. Dieses Verfahren garantiert jedoch *absolut nicht*, dass die berechnete Tweet-ID auch tatsächlich zu dem gewünschten Zeitpunkt gehört. Vielmehr ist die berechnete Tweet-ID eine *sehr grobe* Schätzung des gewünschten Zeitpunkts.

Das Verfahren ist also in der Theorie bereits nicht besonders vielversprechend. Mit dem dazu verbundenen hohen Implementierungsaufwand sowie einer Umstellung des Datenbankschemas wurde dieser Ansatz also verworfen, da er nicht vernünftig in kurzer Zeit realisierbar gewesen wäre.

## Profile

Zum jetzigen Zeitpunkt ist es nicht möglich, weitere Twitter-Profile zur Laufzeit hinzuzufügen oder zu entfernen. Um dies zu erreichen, muss der Daemon explizit beendet und neu gestartet werden. Das ist jedoch umständlich und kostet unnötig Zeit. Das erwartete Verhalten ist vielmehr, dass der Daemon automatisch erkennt, wenn ein neues Twitter-Profil im entsprechenden Verzeichnis vorliegt oder plötzlich fehlt. Entsprechend sollte der Daemon in diesen Fällen dann darauf reagieren.

Aus Zeitgründen und der Tatsache, dass ein solches Verhalten lediglich eine Komforteigenschaft ist, wurde jedoch auf eine Implementierung verzichtet.

## Logging

Der Daemon nutzt für das Festhalten von Fehlern, Warnungen oder anderen Informationen ein minimalistisches Logging-System, welches selbst implementiert wurde. Auf die Nutzung einer Logging-Bibliothek wurde verzichtet. Der Entschluss für den Verzicht wurde wieder aufgrund der Zeitbegrenzung getroffen. Das selbst implementierte Logging-System unterstützt neben der totalen Basis des Loggings einzelnen Meldungen nahezu keinerlei weitere Funktionalität. Lediglich die Verwendung der Log-Datei ähnlich zu einem Ringpuffer wird noch unterstützt. Das bedeutet, dass die entsprechende Log-Datei nicht mehr Zeilen als spezifiziert haben soll. Tritt dieser Fall ein, so werden die ältesten Log-Einträge gelöscht, da diese vermutlich keine bis wenig Relevanz mehr haben.

Der Entschluss gegen die Verwendung einer robusteren und im Funktionsumfang größeren Logging-Bibliothek fiel, da sowohl die Verwendung der von Java selbst bereitgestellten Klassen als auch der Bibliothek Log4J komplex und schwierig ist und viel Entwicklungszeit benötigt hätte [6]. Es war daher schneller, ein eigenes minimalistisches Logging-System zu entwickeln, das unsere Anforderungen bezüglich des Loggings erfüllt. Eine nachträgliche Integration einer externen Logging-Bibliothek in den Daemon und das Entfernen des aktuellen Loggings sollten allerdings nicht zu aufwändig sein, sofern der Entwickler gut mit der externen Bibliothek vertraut ist und genau weiß, wie sie zu verwenden ist.

## Sentiment-Auslagerung

Das Sentiment (siehe dazu Abschnitt 5.1) von Tweets wird momentan im Daemon berechnet. Hierbei ergeben sich jedoch ein paar Probleme. Zum einen kann es zu inkonsistenten Sentiment-Einträgen in der Datenbank kommen, falls das Sentiment gerade aktualisiert wird und der Daemon unerwartet beendet wird oder abstürzt. Dieses unerwartete Verhalten möchte man vermeiden. Des Weiteren benötigt das Sentiment zeitweise sehr viele Ressourcen was Speicherverbrauch und CPU-Belastung betrifft. Zudem wird die Datenbank – welche permanent mit neuen Tweets befüllt wird – durch das Sentiment-Update belastet, weswegen das Abspeichern neuer Tweets massiv verlangsamt und so die Effizienz des Daemons negativ beeinflusst wird.

Aus diesen Gründen ist es eine Überlegung wert, das Sentiment in ein eigenes Programm auszulagern und völlig vom Daemon zu entkoppeln. Dies bringt mehrere Vorteile mit sich. So beeinflusst das Beenden des Daemons das Sentiment nicht, weswegen es aus

diesem Grund auch nicht mehr zu inkonsistenten Sentiment-Einträgen in der Datenbank kommen kann. Ein Absturz des Sentiment-Prozesses, der dann zu inkonsistenten Werten sorgt, ist allerdings weiterhin möglich. Ein weiterer Vorteil der Auslagerung ist die erhöhte Performance des Daemons zu Zeiten der Sentiment-Aktivität, da beide Aspekte – die Suche nach Tweets und das Updaten des Sentiments – nicht mehr um die gemeinsamen Ressourcen konkurrieren. Stattdessen hat jeder Prozess seine eigenen Ressourcen zur Verfügung, die optimal ausgenutzt werden können. Ebenfalls kann nun extern die Priorität der einzelnen Programme, Daemon und Sentiment, geregelt werden. Hierbei ist beispielsweise eine niedrige Priorität für den Sentiment-Prozess sinnvoll, da Änderungen innerhalb der Datenbank nicht sofort verfügbar sein müssen, sondern über einen längeren Zeitraum eingepflegt werden sollen. Zudem lässt sich auch innerhalb der Datenbank die Priorität regeln. So ist es wünschenswert, dass der REST-Service die höchste Priorität beim Arbeiten mit der Datenbank hat, da der Benutzer möglichst schnell Antworten erhalten möchte. Der Daemon sollte niedriger priorisiert sein, um den REST-Service nicht zu behindern. Allerdings sollte die Priorität vom Daemon immer noch höher als die vom Sentiment-Prozess sein, um die zuvor genannten Effizienz nicht negativ zu beeinflussen.

Ein Nachteil der Auslagerung ist, dass das Sentiment für neue Tweets nicht mehr sofort bestimmt wird. Da das Sentiment bislang im Daemon angesiedelt ist, kann, während ein neuer Tweet in die Datenbank eingepflegt wird, auch sofort das Sentiment für diesen Tweet bestimmt und mit abgespeichert werden. Eine Auslagerung in ein eigenes Programm würde nun bedeuten, dass das Sentiment für neue Tweets zunächst unbestimmt ist und erst zu einem späteren Zeitpunkt berechnet werden kann.

Die Vorteile eine Auslagerung des Sentiments aus dem Daemon überwiegen jedoch gegenüber den Nachteilen, weswegen das Trennen beider Komponenten sinnvoll wäre.

## Speicherverbrauch

Während der letzten Iteration des Projektseminars traten leider massive Speicherprobleme innerhalb des Daemons auf, die das Programm häufig abstürzen schließen und so einen Produktiveinsatz unmöglich machten. Aus diesem Grund wurden daraufhin einige Speicheroptimierungen durchgeführt, die das Problem behoben haben. Dennoch ist die Verwendung des Speichers innerhalb des Daemons immer noch nicht optimal und ließe sich sicherlich an einigen Stellen verbessern. Aus Zeitgründen – primär aber, weil das Problem erst viel zu spät bekannt wurde – konnte neben den rudimentären Optimierungen hierauf jedoch nicht weiter eingegangen werden.

## Daemon-API

Es gibt zur Zeit keine komfortable Möglichkeit, Informationen über den Daemon abzufragen. Um zu wissen, was der Daemon gerade macht, muss in die entsprechende Log-Datei geschaut werden und jeder einzelne Log-Eintrag analysiert werden. Das ist, insbesondere für einen Menschen, sehr mühselig und fehleranfällig. Ebenfalls fehlt in der Log-Datei ein großer Anteil an Informationen, die für eine Log-Datei unnötig erscheinen, dennoch interessant zu wissen wären. Es wäre daher ratsam gewesen, eine Daemon-API zu entwickeln, sodass beispielsweise das Frontend mit dem Daemon kommunizieren kann. Dieses könnte dann anzeigen, nach welchen Suchbegriffen in diesem Moment mit welchem Twitter-Profil gesucht wird, welche Profile gerade ihr *rate limit* ausgeschöpft haben, ob der **Treasurer** aktiv ist, wie viele **Treasurer** warten etc. Mit solch einer API könnte über den Daemon alles in Erfahrung gebracht werden.

Ebenfalls könnte hierdurch die Architektur bezüglich des *Royal Treasurers* gegebenenfalls wegfallen. Denn der Daemon könnte dem REST-Service die zeitnah gefundenen Tweets direkt mitteilen und müsste nicht den „Umweg“ über die Datenbank gehen. Dies könnte auch sehr viel schneller sein, da keine Datenbankzugriffe nötig sind. Die Tweets selbst würden dann im normalen Beutel des **Masters** temporär gespeichert werden und zu einem späteren Zeitpunkt in der Datenbank eingetragen werden.

Da die Daten unter anderem auch lokal gecached werden (siehe hierzu den Abschnitt Service-Klassen in 4.3.4 und den Abschnitt 4.4.4 zu Performance-Optimierungen des Servers), würde eine erneute Anfrage zu diesem Suchbegriff auch keine erneute Suche bei Twitter starten. Problematisch wird es lediglich, wenn der Benutzer seinen lokalen Cache löscht und die Tweets noch nicht in die Datenbank geschrieben wurden. Unter der Annahme, dass es keinen **Royal Treasurer** mehr gibt, wird dann eine neue Suche bei Twitter gestartet, die wieder einige Anfragen des besonderen Twitter-Profiles verbraucht und genau dieselben Tweets liefert, die bereits im Beutel des **Masters** liegen. Das ist zwar unschön, allerdings nicht besonders tragisch und könnte akzeptiert werden.

Da eine Anzeige der Informationen über den Zustand des Daemon im Frontend und gegebenenfalls weitere Features aufgrund der Daemon-API jedoch nicht vom Kunden gewünscht wurden und die verfügbare Zeit stark eingeschränkt war, kam es nicht zur Entwicklung einer Daemon-API.

## 4.2. Datenbank

Wie im Kapitel Daemon 4.1 schon angedeutet, ist der Grund für das Verwenden einer Datenbank einerseits den Beschränkungen der Twitter-API geschuldet, andererseits erlaubt uns die Verwendung einer eigenen Datenbank, Twitter-Daten zu erweitern (z. B. durch Sentimentwerte). Wie in Kapitel 2.3 erwähnt, wird in unserem Projekt die Datenbank MySQL verwendet.

### 4.2.1. Schema

Unser Anspruch ist, möglichst jede Information, die Twitter uns liefert, auch selbst zu speichern, da gerade zu Beginn des Projektseminars noch nicht abzusehen war, welche Informationen letztendlich genutzt werden. Deshalb orientiert sich unser Schema sehr stark an der offiziellen Dokumentation von Twitter für Tweets [31] und User [32]. Dabei haben wir mit wenigen Ausnahmen fast alle Werte übernommen, die Twitter zur Verfügung stellt.

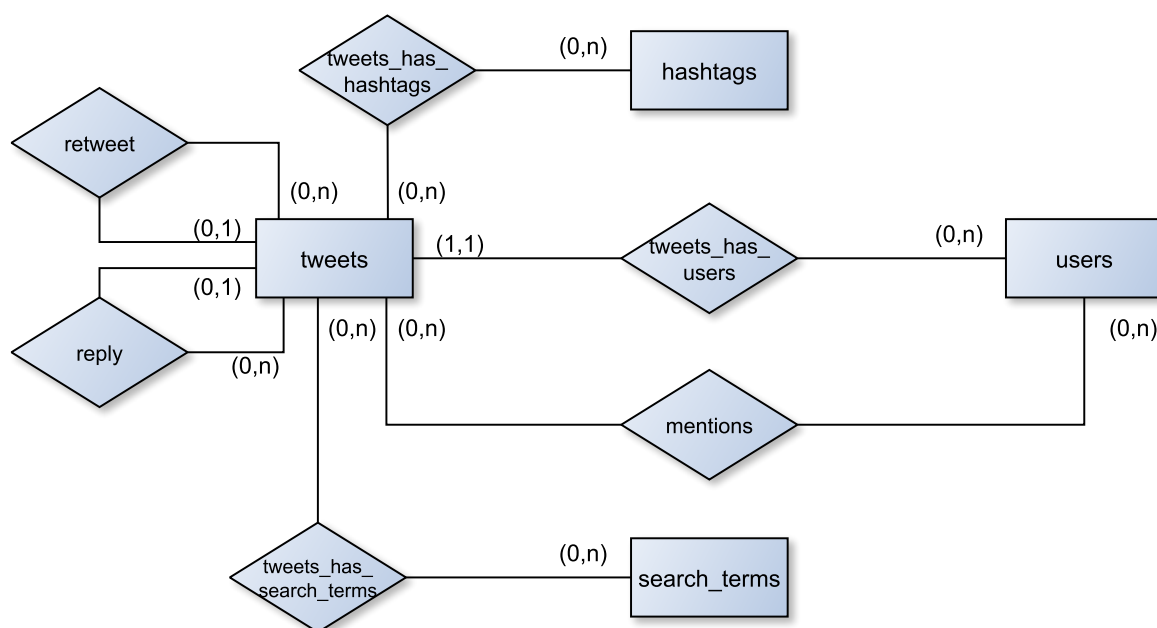


Abbildung 4.6.: Bei Projektbeginn aufgestelltes Entity-Relationship-Modell

Bei der Datenbankmodellierung wurde als erster Schritt ein Entity-Relationship-Modell aufgestellt. Damit lassen sich die zu speichernden Twitter-Informationen zunächst auf ei-



ner konzeptionellen Ebene betrachten, die den Blick auf das Wesentliche lenkt. Abbildung 4.6 zeigt das ER-Modell, welches zu Projektbeginn aufgestellt wurde und trotz späterer Datenbankänderungen nach wie vor die Grundstruktur zeigt. Beispielsweise ist ein Tweet genau einem User zugeordnet, wobei ein User beliebig viele Tweets veröffentlichen kann. Ebenso können in einem Tweet mehrere Nutzer erwähnt sein und andersherum kann ein Nutzer in mehreren Tweets erwähnt sein ( $(0, n) - (0, n)$ -Beziehung **mentions**). Außerdem sind Retweets und Replies als Hierarchie modelliert, da hier eine Beziehung zwischen gleichen Entities entsteht, nämlich zwei Tweets. Zum Beispiel bezieht sich ein Retweet immer auf einen Originaltweet, sodass einem Tweet maximal ein anderer übergeordnet werden darf. Außerdem können einem Tweet beliebig viele Tweets untergeordnet werden (dessen Retweets).

Der einzige Entity-Typ, der sich dabei nicht direkt aus den Daten von Twitter ergibt, ist **search\_terms** mit dem Relationship-Typ **tweets\_has\_search\_terms**. Der Entity-Typ **search\_terms** speichert die vom Nutzer eingegebenen Suchbegriffe, sowie später hinzugefügte Verwaltungsdaten für den Daemon.

Das ER-Modell wurde anschließend in ein relationales Datenbankschema überführt, welches mithilfe von MySQL Workbench erstellt wurde. Dieses Programm erlaubt die direkte Generierung der SQL-Statements, um die entsprechende Datenbank zu erstellen. Den ersten Entwurf des Schemas zeigt Abbildung 4.7.

Zum Beispiel ist zu erkennen, dass die Beziehung **tweets\_has\_users** im ER-Modell nicht in einer eigenen Tabelle resultiert. Stattdessen enthält die **tweets**-Tabelle eine zusätzliche Spalte **users\_id**, die als Fremdschlüssel auf die **users**-Tabelle referenziert. Die beiden Hierarchien **retweet** und **reply** sind auf die gleiche Weise überführt worden. Dagegen resultiert die **mentions**-Beziehung aufgrund der  $(0, n) - (0, n)$  Kardinalitäten in einer eigenen Tabelle.

## Anpassungen des Schema

Im Verlauf der Iterationen gab es mehrere Änderungen am Schema. So gab es zu Beginn Probleme mit der Länge einiger Felder, gerade das Feld für den Text der Tweets war zu kurz, da Twitter die Tweets als UTF-8 kodiert, MySQL aber für UTF-8 mehr als ein Zeichen benötigt. Größere Änderungen brachte der Daemon mit sich. Dieser braucht verschiedene Verwaltungsdaten, die der **search\_terms** Tabelle hinzugefügt wurden. Da die Menge der hinzugefügten Daten recht gering und auch die Anzahl der Einträge der Suchbegriffe insgesamt relativ gering ist, sind dadurch keine negativen Einflüsse auf die

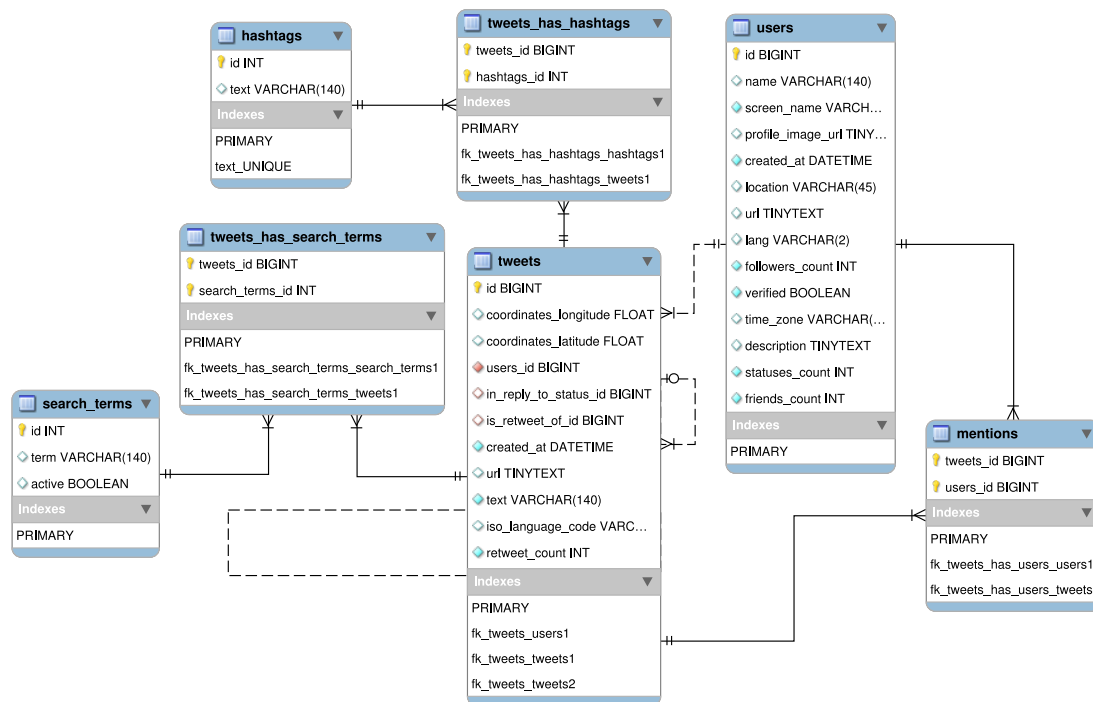


Abbildung 4.7.: Erstes relationale Datenbankschema

Performance zu erwarten. In der letzten Iteration wurde noch eine Denormalisierung des Schemas aus Performance-Gründen eingeführt. Auf das Lösen der Performance-Probleme geht der nächste Abschnitt 4.2.2 genauer ein. Unser endgültiges Schema zeigt Abbildung 4.8. Dabei ist zu sehen, dass die meisten Änderungen, im Vergleich zur ersten Version, an der `search_terms`- und `tweets_has_search_terms`-Tabelle stattgefunden haben.

## Probleme mit UTF-8

Der Text von Tweets, aber auch Selbstbeschreibungen der User und andere Texte sind bei Twitter UTF-8-kodiert. MySQL unterstützt zwar UTF-8, unterscheidet dabei aber zwei Versionen. Die `utf8` genannte Version verwendet dabei nur 3 Byte pro Zeichen und unterstützt deswegen nicht den ganzen Umfang von UTF-8. Daher fehlen z. B. *emoji* und weitere Zeichen, die mit 4 Byte kodiert werden. Der bei MySQL `utf8mb4` genannte Zeichensatz verwendet hingegen 4 Byte pro Zeichen und ist deswegen in der Lage alle Zeichen von UTF-8 darzustellen. Dieser Zeichensatz wird ab MySQL-Version 5.5 unterstützt, während frühere Versionen nur `utf8` unterstützen. Die Unterschiede werden auch in der MySQL-Dokumentation [24] erläutert. Da viele Tweets aus dem asiatischen Raum nicht

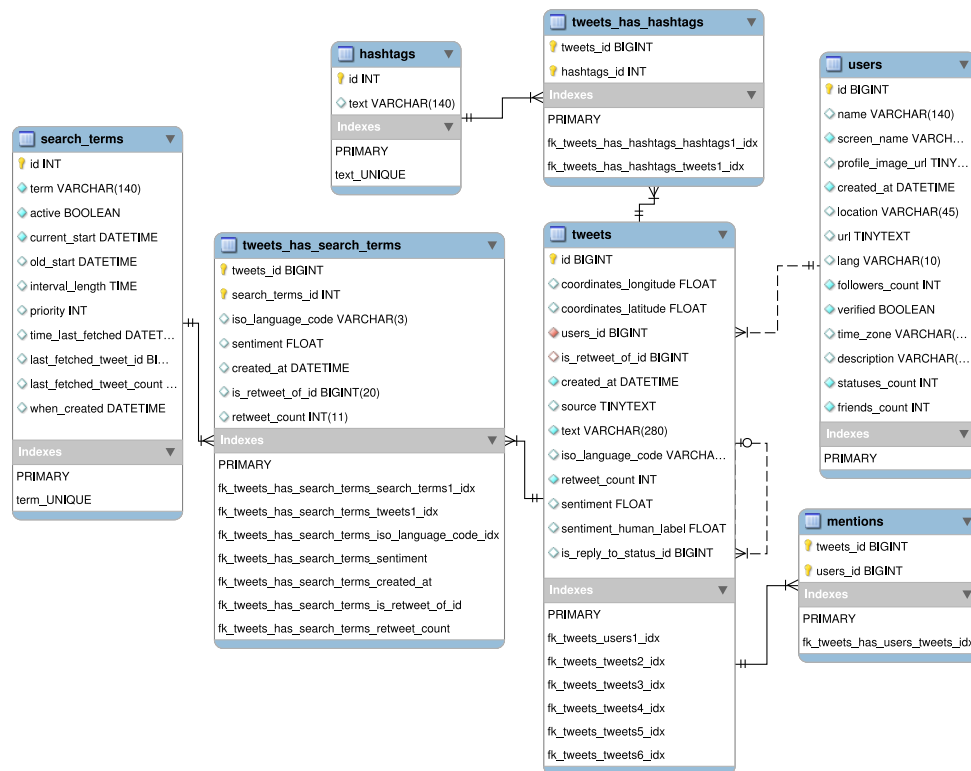


Abbildung 4.8.: Finales relationale Datenbankschema

vom `utf8`-Zeichensatz unterstützte Zeichen und viele *emoji* enthalten, musste MySQL auf Version 5.5 aktualisiert werden und das Schema auf `utf8mb4` umgestellt werden.

## 4.2.2. Performance

Ein großes Thema für uns war die Performance der Datenbank. Sobald der Daemon genug Tweets gesammelt hatte, stellte sich heraus, dass die Performance der Anwendung, insbesondere der Datenbank, nicht unseren Ansprüchen genügte. Deshalb wurde das MySQL-*Slow-Log* aktiviert, um langsame Abfragen zu identifizieren. Dabei stellte sich heraus, dass ein großer Teil unserer Abfragen langsam war und nicht nur vereinzelte. Problematisch war dabei nur die Ausführungszeit der Abfragen selbst, nicht die Zeit, in der *locks* gehalten werden.

### Indizes

Als erster Lösungsansatz wurde für jede Spalte separat untersucht, ob das Nutzen von Indizes sich positiv auf die Laufzeit der Abfrage, welche auf die entsprechende Spalte zu-

greift, auswirkt. In einigen Fällen konnten so erhebliche Steigerungen festgestellt werden, sodass ein entsprechender Index für die Spalte angelegt wurde. So gibt es zum Beispiel eine Abfrage, die einen Wert aus jeder Zeile aufsummiert, welche nicht vom Index profitiert.

## Unterabfragen

Trotz der Indizierung einiger Spalten in der Datenbank hatten einige Abfragen noch eine sehr hohe Laufzeit. Der Grund dafür sind die aufwändigen Joins der Tabellen. Damit weniger Datensätze in einem Join verbunden werden müssen, wurden Unterabfragen eingeführt. So wurden zum Beispiel vor dieser Veränderung zuerst alle Datensätze der `tweets`-Tabelle mit der `tweets_has_search_terms` gejoint und danach erst die Einschränkungen im `where`-Teil der Abfrage ausgewertet. Eine Verwendung von Unterabfragen führt dazu, dass die Einschränkungen vor dem Join ausgewertet werden und nur die relevanten Datensätze mit der `tweets_has_search_terms` verbunden werden. Dadurch wurde zum einen die Zeit verringert, die benötigt wird, um den Join durchzuführen. Ebenfalls wird der Speicherbedarf der Anfrage im RAM verringert. Teilweise konnte dabei dennoch beobachtet werden, dass MySQL die temporären Tabellen auf die Festplatten auslagert, weil sie zu groß für das RAM waren.

## Denormalisierung

Um das Problem der zu großen Tabellen weiter zu lösen, wurde in der letzten Iteration beschlossen, das Schema zu denormalisieren und einige Werte, die ursprünglich ausschließlich in der `tweets`-Tabelle standen, in die `tweets_has_search_terms`-Tabelle zu übernehmen. Hierdurch konnte der Join dieser beiden Tabellen in vielen Abfragen vermieden werden. Dies bewirkte eine weitere Verkleinerung der temporären Tabellen. Die Änderungen sowohl an der Datenbank als auch an der Anwendung waren vergleichsweise klein und schnell umzusetzen und brachten an vielen Stellen bemerkbare Performanceverbesserungen.

## Zählen, Sortieren und weiteres

Aufgrund der Anforderungen der Auswertung, wie zum Beispiel vielen zeitlichen Verläufen, gibt es viele Abfragen, die Felder in den Tabellen zählen und teilweise auch sortieren müssen. Diese Funktionen von MySQL sind aber für die Anzahl an Datensätzen, die in unserer Datenbank vorliegen, ineffizient. Um diesen Flaschenhals zu entfernen, müssten die entsprechenden Werte in einer separaten Tabelle gespeichert werden. Das Speichern

dieser Werte müsste vom Daemon übernommen werden. Hierbei würde zum Beispiel die Anzahl der Tweets zu einem Suchbegriff inkrementell berechnet und abgespeichert. Da solche so genannten *rollup*-Tabellen größere Veränderungen sowohl am Schema als auch am Daemon erfordert hätten, wurde aus Zeitgründen darauf verzichtet.

## Dirty Reads

Ein weiteres Problem war, dass die Ausführungszeit der Abfragen stark abnimmt, sobald der Daemon Tweets in der Datenbank speichert. Da die Abfragen immer erst warten, bis der Daemon die neuen Werte gespeichert hat, konnte dies die Ausführungszeit deutlich erhöhen. Da die Abfrage selbst aber nicht auf die allerneuesten Tweets angewiesen ist, zumal die Tweets, die der Daemon einspeichert ja zu ganzen anderen Suchbegriffen gehören können, wurden die sogenannten *dirty reads* aktiviert. Dabei kann eine Abfrage die Daten lesen, selbst wenn der Daemon momentan weitere Datensätze in die Tabelle schreibt. Dies kann dazu führen, dass die Abfragen nicht alle relevanten Datensätze zurückgeben. Für den Großteil der Anfragen können wir mit diesen gegebenenfalls inkonsistenten Werten aber ohne Probleme arbeiten. Diese Veränderung löst das Problem der Datenbankanfragen, die selbst eine lange Anfragezeit haben, allerdings nicht. Lediglich die Auswirkungen gleichzeitiger Schreibe- und Lese-Operationen können so gemindert werden.

## Fazit

Die Datenbank hat im Laufe des Projektseminars mehrere Iterationen durchlaufen. Zum Ende des Projekts lässt sich zusammenfassen, dass die Performance weiter gesteigert werden muss, um insbesondere den Anforderungen einer Echtzeitanwendung gerecht zu werden. Die Veränderungen haben bereits deutliche Verbesserungen zur Folge gehabt. Ein möglicher nächster Schritt wäre die weitere De- und anschließend erneute Normalisierung des Schemas, um zum Beispiel die oft abgefragten von den selten abgefragten Daten zu trennen, oder die Einführung der oben erwähnten *rollup*-Tabellen für Metadaten, wie Summen von Tweets.

## 4.3. REST

### 4.3.1. Einführung

Das Wort REST ist eine Abkürzung für *Representational State Transfer* und bezeichnet ein Programmierparadigma speziell für Webanwendungen. Vorgeschlagen wurde es von Roy Fielding in seiner Dissertation aus dem Jahr 2000 [15]. In einem Satz lässt sich das Paradigma wie folgt zusammenfassen: REST sieht vor, dass bei einer Anfrage an eine URL immer die gleichen Ressourcen vom Server zurückgeliefert werden. Das Programmierparadigma lässt sich am einfachsten anhand seiner Eigenschaften beschreiben.

#### Client-Server

Die Kommunikation zwischen dem Server und dem Webbrowser oder Client findet nicht direkt statt, sondern ist durch ein Interface getrennt. Dies bedeutet, dass die beiden Komponenten keinerlei Informationen über den Zustand des anderen benötigen und sich nicht um Implementierungsdetails der anderen Komponente kümmern müssen. Dies ermöglicht eine einfache Portierung der Client-Software über verschiedene Plattformen hinweg. Die Skalierbarkeit des Servers wird ebenfalls verbessert, da seine Komponenten vereinfacht werden können. Der größte Vorteil dieser Eigenschaft ist allerdings die Möglichkeit, Server und Client getrennt voneinander zu entwickeln, sofern vorher eine Schnittstelle definiert wurde.

#### Zustandslosigkeit

Eine zusätzliche Einschränkung, die zu der Client-Server Kommunikation hinzugefügt wird, ist die Bedingung, dass die Kommunikation zustandslos sein muss. Dies bedingt, dass alle relevanten Informationen für eine Anfrage an den Server in dieser enthalten sein müssen und die Session auf der Clientseite abgehandelt werden muss. Diese Bedingung verbessert ebenfalls die Skalierbarkeit, da zwischen den einzelnen Anfragen keine zusätzlichen Informationen gespeichert werden müssen. Weiterhin wird die Robustheit des Systems gestärkt, da ein Fehler innerhalb einer Anfragebearbeitung keine weiteren Anfragen beeinflusst.

## Cache

Um die Ressourcen des Netzwerks optimal zu nutzen wird vorausgesetzt, dass Anfragen an den Server explizit als *cacheable* markiert werden müssen.

## Einheitliches Interface

Die Interfaces, die die Kommunikation zwischen verschiedenen Komponenten des Programms abwickeln, müssen einheitlich sein. Hierdurch wird die allgemeine Systemarchitektur vereinfacht und die Kommunikation bleibt einheitlich.

## Schichten

Die Struktur des Programms sollte in Schichten unterteilt sein, wobei keine Schicht mit einer anderen interagiert, die nicht ihre direkte Nachbarschicht ist. So ist es möglich, bestimmte Funktionen in einer Schicht zu gruppieren und dadurch die Komplexität des Programms weiter zu reduzieren.

Da unser Endprodukt eine Webanwendung für verschiedene Plattformen sein soll, erschien dem Team dieses Programmierparadigma als sinnvoller Ansatz für unsere Server-Architektur.

### 4.3.2. Implementierung

Zur Implementierung unseres REST-Services in Java verwenden wir Jersey [5], die Standardbibliothek für Java-APIs für RESTful Services (JAX-RS). Jersey ermöglicht die Implementierung eines REST-Services innerhalb einer Java-Klasse, ohne die dabei notwendige HTML-Kommunikation explizit behandeln zu können. Wie bereits beschrieben, sind in einem REST-Service serverseitige Ressourcen definiert, mit denen ein Benutzer als Client über HTML-Anfragen an eine assoziierte URL interagieren kann.

Ein REST-Service funktioniert also immer in Reaktion auf eine Benutzer-Anfrage. Um diese Reaktion in Java implementieren zu können, kann in Jersey eine Ressource mit einer Java-Methode in Verbindung gesetzt werden. Das geschieht, indem für die Methode spezifiziert wird, welche Art von HTML-Anfrage auf welche URL zum Aufruf der Methode führt. Die URL setzt sich dabei aus den Einstellungen für die globale Basis-URL (welche gemeinsam mit anderen Einstellungen in einer `web.xml`-Datei festgelegt werden kann), dem URL-Bestandteil der Klasse und dem URL-Bestandteil der Methode zusammen. Die

Bestandteile der Klasse und der Methode können dabei durch den Parameter der von Jersey bereitgestellten Annotation `@Path` definiert werden. Zusätzlich stehen außerdem Annotationen zur Verfügung, welche die Art der HTML-Anfrage spezifizieren, da jeweils unterschiedliche Methoden für ihre Ausführung notwendig sind. Von diesen haben wir ausschließlich `@GET` und `@POST` verwendet. Eine beispielhafte Verbindung der Methode `getTweets()` GET-Anfrage auf die URL `www.tmetrics.de/rest/tweets` (zusammengesetzt aus der Basis-URL `www.tmetrics.de` sowie den Bestandteilen `/rest` der Klasse und `/tweets` der Methode) ist in Abbildung 4.9 zu sehen.

```

17 @Path("/rest")
18 public class Service
19 {
20     @GET
21     @Path("/tweets")
22     public Response getTweets()
23     {
24         return Response.status(HttpURLConnection.HTTP_OK).entity(null).build();
25     }
26 }

```

Abbildung 4.9.: Jersey: Verbindung von Methoden mit HTML-Anfragen

Wie ebenfalls an der Abbildung erkennbar ist, liefert jede mit Jersey implementierte Methode eines REST-Servers standardmäßig eine HTML-Response zurück, welche durch die `Response`-Klasse repräsentiert wird. Die eigentlichen Antwortdaten können dabei als Bestandteil des Entity-Attributs der Response übergeben werden (in diesem Beispiel nur `null`). Dabei kann mit einer weiteren Annotation spezifiziert werden, in welcher Form die Daten in der Entity an das Frontend gesendet werden. Da das Frontend in JavaScript implementiert ist, haben wir uns hier für das auf JavaScript ausgerichtete JSON-Format entschieden. Dieses kann über den Parameter der Annotation `@Produces` spezifiziert werden (siehe Abbildung 4.10). Primitive Datentypen sowie in JavaScript vorhandene Datenstrukturen werden dabei automatisch geparkt. Für eigene Klassen muss das Parsing explizit definiert werden (siehe Abschnitt 4.3.3). Dieses Parsing macht es sogar möglich, anstatt einer expliziten Response direkt die Daten zurückzugeben. Diese werden dann implizit in eine Response verpackt. Dieses Vorgehen wurde zeitweise von uns eingesetzt, aber im Projektverlauf verworfen, da uns eine explizite Kontrolle der Response insbesondere wegen der Statusnachricht als sinnvoll erschien.

Häufig ist außerdem der Zugriff nur auf bestimmte Bestandteile einer Ressource notwendig. So ist niemals der gesamte Inhalt der Tweets-Ressource von Interesse, sondern nur die Tweets, die mit einem bestimmten Suchbegriff verbunden sind (welcher durch



```

19 @GET
20 @Path("/tweets")
21 @Produces(APPLICATION_JSON)
22 public Response getTweets()
23 {
24     List<Tweet> tweets = new ArrayList<Tweet>();
25
26     return Response.status(HttpURLConnection.HTTP_OK).entity(tweets).build();
27 }

```

Abbildung 4.10.: Jersey: Spezifizierung des Rückgabeformats

eine ID identifiziert wird). Es sind außerdem weitere Einschränkungen der Daten vorstellbar. So sind in bestimmten Ansichten nur Tweets aus einem bestimmtem Zeitraum oder mit einem bestimmten Sentiment von Interesse. Diese Parameter finden sich einerseits als Parameter in der URL der HTML-Anfrage und andererseits in der Signatur der entsprechenden Java-Methode wieder. Die Jersey-Annotation `@QueryParam` ermöglicht dabei die Verbindung eines Parameters der Anfrage mit einem Argument in der Methodensignatur. Dabei ist zu beachten, dass sämtliche Argumente `null` sind, falls ihnen entweder im Quellcode kein Anfrageparameter zugewiesen wurde oder falls die Anfrage diesen Parameter nicht beinhaltet. Soll also ein Parameter verpflichtend gemacht werden, muss diese Situation abgefangen und eine Exception geworfen werden. Eine Alternative bildet die Verwendung der Annotation `@DefaultValue`, womit ein Standardwert festgelegt wird, falls der Parameter nicht in der Anfrage vorkommt. Eine genauere Behandlung dieser Fälle wird in Abschnitt 4.3.4 beschrieben. Auf ähnliche Weise können Methodenargumente über die Annotation `@HeaderParam` auch auf die Header-Parameter der HTML-Anfrage zugreifen. Diese Fälle werden in Abbildung 4.11 veranschaulicht. Ein valider Zugriff auf die in dieser Methode spezifizierte Ressource wäre beispielsweise eine GET-Anfrage auf `www.tmetrics.de/rest/tweets?id=1&limit=100`. Aufgrund des Standardwertes könnte der `limit`-Parameter dabei auch weggelassen werden.

```

22 @GET
23 @Path("/tweets")
24 @Produces(APPLICATION_JSON)
25 public Response getTweets(@QueryParam("id") final int searchTermID,
26                           @DefaultValue("100") @QueryParam("limit") final int limit,
27                           @HeaderParam("If-Modified-Since") final String modified)
28 {
29     List<Tweet> tweets = getTweetsFromDB(searchTermID, limit);
30
31     return Response.status(HttpURLConnection.HTTP_OK).entity(tweets).build();
32 }

```

Abbildung 4.11.: Jersey: Übergabe von Parametern

Auf diese Weise ermöglicht Jersey es, Java-Methoden mit HTML-Anfragen auf eine URL zu verknüpfen und die Eingaben und Ausgaben dieser Methode zu kontrollieren. Um eine konkrete HTML-Anfrage abzuhandeln, wird die Klasse, in der Jersey implementiert ist, neu instanziiert und die korrekte Methode identifiziert und ausgeführt. Diese neuerliche Instanziierung bei jeder Anfrage spiegelt die Tatsache wieder, dass ein REST-Service zustandslos ist.

### 4.3.3. Struktur des REST-Service

#### Klassenstruktur

Wie bereits erwähnt, ist die Aufgabe unseres REST-Service, eine Verbindung zwischen Benutzeranfragen und -eingaben auf der einen Seite und der Datenbank auf der anderen Seite herzustellen. Die Schnittstelle zur HTML-Kommunikation mit dem Benutzer bildet den REST-Service im engeren Sinne. Dieser wird bei uns von einer Klasse **Service** übernommen, in der wie zuvor beschrieben mit Jersey die REST-Schnittstelle implementiert wird. Die Kommunikation mit der Datenbank findet in der Klasse **Transactor** mittels JDBC statt.

Dabei gibt es jedoch keine direkte Kommunikation zwischen den Klassen **Service** und **Transactor**. Denn wie bereits in der Architekturübersicht dargestellt, besteht die Aufgabe des REST-Service im weiteren Sinne nicht nur in der Ausgabe von Daten aus der Datenbank, sondern auch aus deren Analyse. Zu diesem Analyseverfahren gehören das Clustering, die Identifikation von Peaks und die Darstellung von *related news* sowie das Bestimmen von Einflussfaktoren des Sentiment-Wertes von Tweets. Damit diese Verfahren integriert werden können, befindet sich zwischen den **Service**- und **Transactor**-Klassen noch eine weitere Klasse **Logic**. Der Fluss der Daten sieht dabei folgendermaßen aus: eine REST-Anfrage des Benutzers ruft über Jersey die zugeordnete Methode der **Service**-Klasse auf. Diese Anfrage wird zunächst über die entsprechende Methode in **Logic** an eine **Transactor**-Methode weitergereicht, welche die benötigten Daten aus der Datenbank abrufen. Diese werden dann an die **Logic**-Methode zurückgeliefert. Beinhaltet die Anfrage den Einsatz eines Analyseverfahrens, werden diese durch Einsatz der jeweils damit verbundenen Klassen an dieser Stelle durchgeführt. Wird beispielsweise das Clustering angefragt, liefert der **Transactor** eine entsprechende Auswahl Tweets, auf der dann in der **Logic**-Methode das Clustering durchgeführt wird.

Ein weiterer Aufgabenbereich der **Logic**-Klasse ist in einigen Fällen die Aufbereitung der vom **Transactor** zurückgelieferten Daten. Um beispielsweise den zeitlichen Verlauf der Aktivität oder des Meinungsbildes zu erhalten, werden von der Datenbank sämtliche Tweets nach der Stunde ihrer Erstellung gruppiert. Die Methode im Frontend, die diese Daten schlussendlich darstellen soll, erwartet dabei Paare aus Zeitpunkten und einer Anzahl in konstanten Intervallen. Da die Rückgabe der Datenbank die vollen Stunden, an denen gar keine Tweets erstellt wurden, gar nicht beinhaltet, müssen in der **Logic**-Klasse diese Zeitpunkte mit der Anzahl 0 nachträglich eingefügt werden.

Ist weder eine Analyse noch eine Aufbereitung der Daten notwendig, haben wir uns aus Gründen der Konsistenz der Implementierung entschieden, die **Logic**-Klasse nicht zu umgehen. In diesem Fall reicht sie die Ergebnisse einfach weiter.

Wie bereits im vorigen Kapitel angemerkt wurde, lassen sich mit Jersey sowohl Klassen als auch Methoden mit einem URL-Bestandteil versehen. Um eine semantische Trennung der REST-Anfragen zu gewährleisten, wurden auf diese Weise zwei REST-Service-Klassen implementiert. Die Klasse **ResultService** mit dem Pfadbestandteil `/results/` ist dabei für sämtliche Anfragen zuständig, welche mit den direkt für den Benutzer im Frontend sichtbaren Resultaten zu tun haben, d. h. den Inhalten der Views. Die Klasse **QueryService** mit dem Pfadbestandteil `/queries/` ist hingegen für alle übrigen internen Anfragen verantwortlich, wie zum Beispiel die Anfrage, ob ein spezifischer Suchbegriff bereits in der Datenbank vorhanden ist oder welche ID dieser hat. Analog dazu existieren die zwei **Logic**-Klassen **ResultLogic** und **QueryLogic**, welche allerdings weiterhin auf eine gemeinsame **Transactor**-Klasse zugreifen. Hintergrund dieser Trennung ist eine übersichtlichere Struktur sowohl der REST-API als auch des internen Java-Codes, welcher aufgrund der Vielzahl der Methoden ansonsten umständlich zu handhaben gewesen wäre.

## Data Transfer Objects

Um die Kommunikation sowohl innerhalb des REST-Service als auch mit dem Frontend zu vereinfachen, haben wir uns für den Einsatz von *Data Transfer Objects* (DTOs) entschieden. Dabei handelt es sich um ein *Plain Old Java Object* (POJO), welches nur über private Felder sowie deren zugeordnete Getter und Setter verfügt. Da für gewöhnlich die Rückgabe einer Datenbankanfrage im Transactor sehr komplexe Daten mit einer großen Menge von Attributen zurückgibt, ist es sinnvoll, diese Daten in einem DTO zu kapseln, sodass sie leicht zu manipulieren sind und einfach zwischen Methoden verschickt werden können. Ein weiterer Vorteil ist, dass sich die DTOs unter Einsatz eines POJO-Mappers

implizit ins JSON-Format parsen lassen, welches sich einfach in der JavaScript-Umgebung des Frontends verwenden und in Jersey als Rückgabebetyp einer REST-Anfrage spezifizieren lässt.

Dies geschieht in unserem Projekt mithilfe der Bibliothek Jackson [3]. Diese stellt eine Reihe von Annotationen bereit, welche bei der Implementierung eines DTOs verwendet werden können, um festzulegen, wie dessen Attribute geparkt werden sollen. Am wichtigsten sind dabei hier die Annotationen `@JsonProperty`, dessen Parameter festlegt, unter welchem Namen das Attribut im JSON geparkt werden soll, sowie `@JsonIgnore`, das bestimmt, dass dieses Attribut beim Parsen ignoriert werden soll. Dabei können sämtliche primitiven Datentypen sowie alle Klassen, für die ebenfalls ein Jackson-Parsing definiert wurde, mit `@JsonProperty` versehen werden. Des Weiteren ist Jackson in der Lage, Arrays sowie Datenstrukturen mit direkter Entsprechung in JavaScript wie z. B. `List` oder `Map` in ihr JSON-Gegenstück zu parsen. Ein so über Jackson konfiguriertes DTO kann dann direkt im REST-Service als implizite HTML-Response an das Frontend zurückgeliefert werden, wo es im JSON-Format ankommt.

Auf diese Weise wurde für sämtliche Ressourcen ein geeignetes DTO implementiert. In einigen Fällen können dabei auch mehrere DTOs in einander geschachtelt sein. So sind zum Beispiel für die Darstellung eines Tweets im Frontend auch Informationen über den Benutzer notwendig, der diesen Tweet erstellt hat. Daher existiert ein DTO `TweetWithUser`, das wiederum die DTOs `Tweet` und `User` beinhaltet. Zusätzlich zu dieser Schachtelung von DTOs haben wir uns bei der Konzeption dafür entschieden, das Meta-DTO `Envelope` einzuführen, das sämtliche anderen DTOs enthält. Der Vorteil eines solchen `Envelope` ist, dass es die Handhabung von Fehlern und anderem unerwarteten Verhalten ermöglicht. Zu diesem Zweck hat die Klasse neben dem Attribut `data` vom Typ `Object` zum Speichern des jeweiligen DTOs ebenfalls die Felder `status` und `error_codes`. In `status` kann dabei der Erfolg der Operation abgespeichert werden (wobei in Anlehnung an HTML hauptsächlich 200 `OK` für erfolgreiche Operationen und 500 `Internal Server Error` für Fehler verwendet wurden), während `error_codes` es ermöglicht, gegebenenfalls Informationen über die Fehlerursache wie z. B. die Stacktrace einer Exception zu hinterlegen. Da das `data`-Feld auch auf `null` belassen werden kann, kann in so einer Situation selbst dann ein `Envelope` zurückgeliefert werden, falls infolge eines Fehlers keine Daten vorliegen. Der Status sowie die Fehlermeldung ermöglichen dann das Abfangen von unerwartetem Verhalten im Frontend und erleichtern die anschließende Fehlerbehandlung.

Das Verpacken der DTOs geschieht dabei ebenfalls in der **Logic**-Klasse. Das bedeutet, dass das **ResultSet** der SQL-Anfrage im **Transactor** zunächst in einem DTO gekapselt wird, welches an die Methode in **Logic** übergeben wird. Unter Umständen geworfene Exceptions werden dabei ebenfalls zunächst unbehandelt weitergereicht. Wird in **Logic** eine Exception gefangen, liegen für gewöhnlich keine Daten vor, es wird dann ein **Envelope** mit negativem Status und der entsprechenden Stacktrace erstellt. Ansonsten wird das erhaltene DTO zusammen mit einem positiven Status in einen **Envelope** verpackt.

Die hier vorgestellte Struktur wird in Abbildung 4.12 visualisiert.

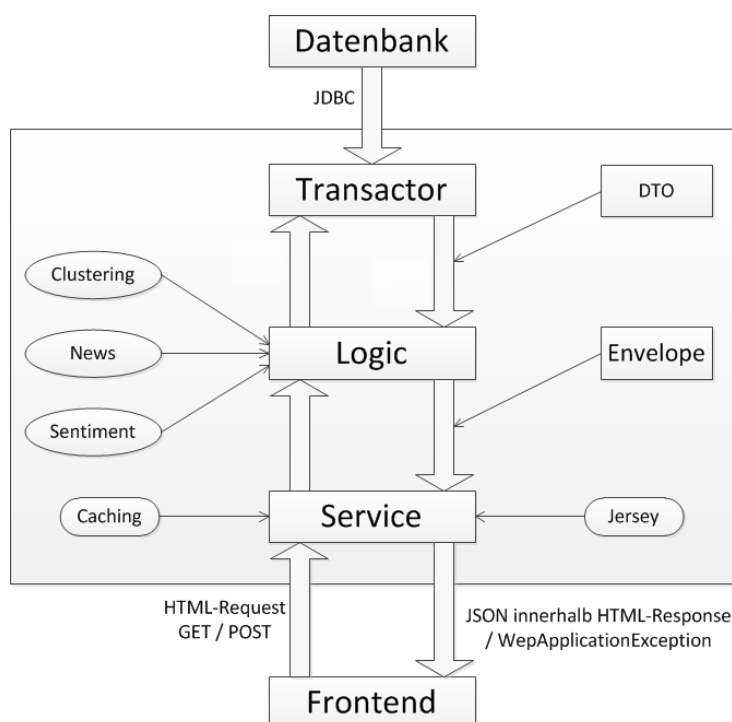


Abbildung 4.12.: Struktur des RestService

#### 4.3.4. Umsetzung und Problemstellungen

##### Transactor

Wie bereits oben beschrieben wird im **Transactor** die Verbindung zur Datenbank erstellt und die einzelnen Abfragen abgeschickt. Hierzu wird im Konstruktor der Klasse eine Verbindung zur Datenbank hergestellt. Die benötigten Parameter für die Verbindung werden aus einer **properties**-Datei geladen. Für jede Anfrage, die an den REST-Service gesen-

det wird, gibt es in dieser Klasse eine eigene Methode, die die SQL-Anfrage erstellt und abschickt.

Die Anfragen werden mit Hilfe von `PreparedStatement`s erstellt. Ein Anfragestring wird hierbei an den Konstruktor des `PreparedStatement`s übergeben und die einzelnen Parameter mit einem Fragezeichen markiert. Die Werte für diese Parameter können dann im nächsten Schritt anhand eines Indexes gesetzt werden. Bei Anfragen, die optionale Parameter enthalten, müssen diese Anfragestrings dynamisch angepasst werden. Ein Problem, das dabei auftrat, sind die unterschiedlichen Indizes, die der gleiche Parameter bei unterschiedlichen Anzahlen von Anfrageparametern annehmen kann. Ein erster Ansatz dies zu lösen bestand darin, alle verschiedenen Kombinationen von gesetzten und nicht gesetzten Parametern durchzugehen und die Parameter für das `PreparedStatement` explizit zu setzen. Dies ist jedoch bereits ab einer kleinen Anzahl von Parametern nicht mehr praktikabel. Als sehr gute Lösung erwies es sich eine Laufvariable einzuführen, die nach jedem gesetzten Parameter um eins erhöht wird. So musste im Gegensatz zu allen Kombinationen von gesetzten und nicht gesetzten Parametern nur jeder einzelne Parameter überprüft werden. Sobald das `PreparedStatement` fertig gestellt ist, kann es an die Datenbank geschickt werden.

Das Ergebnis der Anfrage liegt dann zuerst in einem `ResultSet` des `java.sql`-Packages vor. Diese `ResultSets` werden dann direkt in die entsprechenden *Data Transfer Objects* umgewandelt. Bei komplexeren DTOs wurden für diese Aufgabe Builder-Klassen implementiert, die das Befüllen der Objekte komfortabler machen.

In einer ersten Version des `Transactors` wurden all diese Operationen in einem `try-catch`-Block ausgeführt, um Exceptions, die bei dem Verbindungsaufbau mit der Datenbank oder beim Ausführen der Anfragen geworfen werden können, abzufangen. In dieser Version wurden die Fehlermeldungen direkt in den `Envelope` geschrieben und dieser zurückgegeben. Da diese Art der Fehlerbehandlung zu Problemen führte, wie im Unterkapitel zu den Service-Klassen näher beschrieben ist, werden die Exceptions in der aktuellen Version direkt weitergeworfen. Die `try-catch`-Struktur wurde in der `Transactor`-Klasse allerdings beibehalten, da dies einen `finally`-Block ermöglicht, in der sämtliche Verbindungen zur Datenbank definitiv geschlossen werden können. Dieses Kappen der Verbindungen ist sehr wichtig, um die Datenbank nicht mit zu vielen offenen Verbindungen zu blockieren.

## Logic-Klassen

In den **Logic**-Klassen werden wie vorher beschrieben die Ergebnisse der Datenbankabfragen weiterverarbeitet. Neben den bisher beschriebenen Aufbereitungen werden hier auch die anderen Funktionen des REST-Service aufgerufen. Die Details der anderen Module, wie Clustern und der Newsanfrage, finden sich in den entsprechenden Komponenten-Kapiteln.

Eine Besonderheit, die hier im Vordergrund stehen soll, ist die Einbindung der Daten für das Sentiment-Modell. Diese Informationen sind notwendig, um die Darstellung der Einflussfaktoren für die Sentimentanalyse eines Tweets zu ermöglichen. Da die Analyse im Daemon stattfindet und die Parameter, die zu dem Ergebnis geführt haben, nicht in der Datenbank vorgehalten werden, muss eine Möglichkeit gefunden werden, die Informationen auf einem anderen Weg in den REST-Service zu laden.

Die erste lauffähige Methode beinhaltete, die Sentiment-Klassen des Daemons in den REST-Service zu laden und die Analyse erneut für die aktuell angefragten Tweets durchlaufen zu lassen. Dies beinhaltet wegen der Zustandslosigkeit in jedem Aufruf ein neues Trainieren der Modelle. Diese Art der Redundanz erschien uns allerdings nicht haltbar. Die Methode, die uns das wiederholte Trainieren der Modelle abnahm, beinhaltete, die trainierten Modelle auf der Festplatte zu speichern und nach Bedarf im REST-Service zu laden. Die Modelle werden im Daemon trainiert und auf der Festplatte aktualisiert, sobald sich neue, von uns gelabelte Tweets, in der Datenbank befinden. Wie im Daemon-Kapitel beschrieben, wird der Thread, der diese Aktualisierung durchführt, einmal am Tag gestartet.

Die vorher beschriebene Eigenschaft der Zustandslosigkeit des REST-Service sorgte allerdings für ein weiteres Problem. Da aufgrund dieser Eigenschaft keine Informationen von einer Anfrage an den Rest-Service zur nächsten übertragen werden konnten, bedeutete dies, dass die Modelle für jede Anfrage neu geladen werden mussten. Bei etwa 5-6 MiB pro Modell und Sprache, bedeutet dieses Laden einen deutlichen Flaschenhals bei parallel laufenden Anfragen. Eine kleine Änderung der Klassenvariable für diese Modelle auf **static** sorgte dafür, dass die Modelle nur bei der ersten Anfrage geladen werden und danach weiterhin vorgehalten werden. Da unser Anspruch, möglichst präzise Informationen zu der Sentimentanalyse zu präsentieren, allerdings dadurch nicht mehr voll erfüllt wurde, musste eine Möglichkeit gefunden werden, die Modelle aktuell zu halten.

Der erste naive Ansatz hierzu war es, die Modelle im Speicher mithilfe einer Hashsumme mit den Modellen auf der Festplatte zu vergleichen. Da eine solche Summe allerdings erst dann für die Modelle auf der Festplatte berechnet werden kann, wenn diese in den Speicher geladen wurden, wurde diese Idee sehr schnell verworfen. Stattdessen wurde entschieden in den Modellen selbst ihre Erstellungszeit zu speichern. Diese Information ist sehr leicht auszulesen und konnte dann mit der Erstellungszeit der Dateien auf der Festplatte verglichen werden. Das Besondere bei dieser Methode ist, dass nicht die interne Erstellungszeit des Modells auf der Festplatte als Vergleichswert herangezogen wird, sondern der Zeitstempel im Dateisystem für die Datei. Sollte der Unterschied zwischen diesen beiden Zeitstempeln einen bestimmten Wert überschreiten, so ist das aktuelle Modell veraltet und wird neu geladen. Hierdurch konnte die zu ladende Menge an Daten pro Anfrage drastisch reduziert werden.

Da so ein akzeptabler Weg gefunden ist, die trainierten Modelle zwischen dem Daemon und dem REST-Service auszutauschen, können sowohl die Einflüsse für das Ergebnis der Sentimentanalyse, als auch die wichtigen Trainings-Tweets, die zu diesem Ergebnis geführt haben, ermittelt werden und an die nächste Schicht im REST-Service, die Serviceklassen, weitergeleitet werden.

## Service-Klassen

Die **Service**-Klassen handeln, wie vorher beschrieben, die Kommunikation in Richtung Frontend ab. Neben dieser Kommunikation werden in diesen Klassen ebenfalls die Antworten erstellt, die dann endgültig an das Frontend geschickt werden. Der oben erwähnte **Envelope** wurde in einer früheren Version unseres Programms hier endgültig befüllt oder eventuelle Exceptions gefangen und in die entsprechenden Felder im **Envelope** geschrieben. Diese Informationen beinhalteten den Stacktrace und den Error-Code der gefangenen Exception. Ebenso wurden Informationen zu fehlenden oder fehlerhaften Anfrageparametern hier in einen **Envelope** geschrieben und an das Frontend weitergeleitet.

Diese Herangehensweise an die Fehlerbehandlung hatte allerdings einen großen Nachteil. Der HTTP-Status-Code der Antwort des REST-Service war bei dieser Implementierung immer 200 OK, egal ob ein Fehler im Programm vorlag, ein Parameter fehlerhaft war, oder die Anfrage ohne Probleme durchgelaufen war. Dies bedeutete, dass im Frontend zur Behandlung nicht vorhergesehener Fehler neben dem HTTP-Status der Antwort auch die internen Felder des **Envelopes** überprüft werden mussten. Um diese doppelte Überprüfung zu eliminieren wurde festgelegt, dass die HTTP-Status-Codes abgefangene



Fehler im Programmcode widerspiegeln sollten. Hierzu wurden die Methoden der Klasse so umgeschrieben, dass sie nun explizit eine HTTP-Response zurückgaben. Die bisherigen `Envelopes` wurden dann als `entity` in diesen `Response`-Objekten gespeichert.

Diese Änderung ermöglichte es ebenfalls, `WebApplicationExceptions` für die Fehlerbehandlung zu benutzen. Eventuell geworfene Exceptions in einer tieferen Schicht des REST-Service werden nun hier gefangen und als `WebApplicationException` an das Frontend weitergereicht. Da diese `WebApplicationException` ebenfalls über ein HTTP-Response-Objekt an das Frontend weitergegeben wird, kann erneut der `Envelope` als `entity` gesetzt werden. Das bedeutet, dass der HTTP-Status-Code `500 Internal Server Error` einen Fehler anzeigt, aber die wichtigen Informationen über die Art des Fehlers dennoch am Frontend auslesbar sind. Diese Informationen waren in Entwicklungsprozess sehr hilfreich, da ein Fehler, der bei der Entwicklung des Frontends auftrat, mit sehr genauen Informationen an die Entwickler des REST-Service weitergeleitet werden konnte.

Die Behandlung fehlerhafter Parameter wurde ebenfalls deutlich erleichtert. Das Protokoll der HTTP-Status-Codes beinhaltet einen speziellen Code, um anzuzeigen, dass die Anfrage an den Server fehlerhaft war: `400 Bad Request`. Dieser Code konnte nun verwendet werden, um anzuzeigen, dass notwendige Parameter fehlten oder Parameter einen ungültigen Wert hatten.

Ein weiteres Feature, das durch diese Änderung ermöglicht wurde, ist das Caching der Ergebnisse im Browser. Hierzu wurde eine zusätzliche Methode in die unteren beiden Schichten des REST-Service eingefügt, die überprüft, wann das letzte mal für einen bestimmten Suchbegriff neue Daten von Twitter abgefragt wurden. Fragt ein Browser zum ersten mal nach der aktuellen Ressource, wird diese Information zusätzlich im Header der HTTP-Response unter dem Feld `Last-modified` gespeichert. Diese Information wird bei einer erneuten Anfrage an die gleiche Ressource erneut im Header mitgeschickt und kann dann zu Beginn der Methode ausgelesen und mit dem erneut aus der Datenbank abgefragten Zeitstempel verglichen werden. Handelt es sich um den gleichen Zeitstempel, so genügt es eine HTTP-Response mit dem HTTP-Status-Code `304 Not Modified` an den Browser zurückzusenden. Dieser verwendet dann zur Anzeige die in der letzten Anfrage erhaltenen Daten.

Diese Methode erspart die komplexen Datenbankabfragen, wenn sich der unterliegende Datensatz nicht geändert hat, und ersetzt sie durch eine einfache Anfrage, die über einen Primärschlüssel abgehandelt werden kann.

## 4.4. Server

### 4.4.1. Anforderungen und Grundsteine

Nachdem die ersten allgemeinen Anforderungen an das Projekt durch das Projektteam festgelegt wurden, ergaben sich daraus ebenfalls die ersten konkreten Anforderungen an den nötigen technischen Unterbau. Da jedoch noch nicht alle Anforderungen bis ins Detail ausgearbeitet waren, wurden einige Punkte umgesetzt, die sich letztendlich nicht als zwingend notwendig erwiesen haben. Im folgenden Abschnitt werden die Details der Services auf dem Server und die generelle technische Umgebung beschrieben.

Aufgrund des allgemeinen Wunsches, laufende Services so weit wie möglich frei konfigurieren zu können, um z. B. Datenbank, HTTP, Mail und weitere Server flexibel an evtl. wechselnde Anforderungen anpassen zu können, war schnell klar, dass ein voller root-Zugang definitiv von Vorteil, wenn nicht unabdingbar wäre. Ein freier root-Zugang versprach ebenfalls schnelle Reaktionen und somit die Vermeidung von unnötigen Wartezeiten und Nachfragen bei der IVV (Informations-Verarbeitungs-Versorgungseinheit des Fachbereiches).

Erste Erkundigungen bei der IVV ergaben jedoch schnell, dass die Möglichkeiten hier stark eingeschränkt sein würden, sodass die Alternative, einen unabhängigen Server durch einen externen Anbieter zu mieten, auf der Hand lag. Die Wahl fiel auf den kleinsten vServer des Anbieters Server4You. Da die Leistungswerte bezüglich RAM und CPU des virtuellen Servers mindestens denen eines durch die IVV zur Verfügung gestellten Rechners entsprachen, war damit der erste technische Grundstein des Servers festgelegt. Der zweite Grundstein bestand aus der Wahl des Betriebssystems. Zur Verfügung standen CentOS 6, Debian 6 und Ubuntu 10.04 LTS, jeweils in einer minimalen Variante und einer mit einem Hosting-Tool ausgestatteten. Obwohl sowohl CentOS 6 als auch Debian 6 neueren Datums sind als Ubuntu 10.04 LTS, machten im Team vorherrschende Kenntnisse über Ubuntu diese Wahl ebenfalls relativ einfach. Da keinerlei Erfahrung mit dem gegebenen Hosting-Tool (Plesk) im Team vorhanden war und wir hier nicht künstlich in den Möglichkeiten eingeschränkt werden wollten, fiel die Wahl auf Ubuntu 10.04 LTS in der minimalen Variante.

Ein erster Überblick über die gegebenen Anforderungen schien nicht gegen Ubuntu 10.04 LTS zu sprechen, da für alle relevanten Systemdienste wie z. B. Tomcat, Apache und MySQL in genügend aktuellen Versionen vorzuliegen schienen. Im Ergebnis kann aber festgehalten werden, dass Ubuntu für unser Team definitiv die richtige Wahl war, da hier

keinerlei weitere Einarbeitungszeiten nötig wurden und alle Teammitglieder selbständig und ohne Probleme mit dem System arbeiten konnten.

Da zu diesem Zeitpunkt noch nicht endgültig feststand, wie der Funktionsumfang unseres Projektes exakt aussehen wird, standen noch einige Punkte wie eine Benutzerverwaltung oder Zugänge im Raum, welche im weiteren Verlauf jedoch an Bedeutung verloren. Dies war zunächst jedoch ein weiteres Argument für einen unabhängigen Server, denn nur so war es möglich, ohne große Aufwände ein eigenes und kostenloses Zertifikat zu integrieren um so die verschlüsselte Verbindung zwischen Browser und Server zu gewährleisten. Dies sollte sich im weiteren Verlauf dennoch als ein durchaus nützlicher Pluspunkt herausstellen. Um evtl benötigte Mailadressen für eine Registrierung bei Twitter zur Verfügung zu haben, wurde die Idee begrüßt einen eigenen Mailserver zu betreiben.

#### 4.4.2. Projektname und Grundkonfiguration

Nachdem die Basis des Servers nun also gegeben war, bestand der nächste Schritt darin, die einzelnen Services zu installieren, konfigurieren und unseren Anforderungen anzupassen. Um den Server über einen Namen ansprechen zu können, legte sich das Team relativ schnell zunächst auf `twittermetrics` und später dann auf `tmetrics.de` fest. Der Grund für diesen Wechsel war, dass wir über den Anbieter StartSSL.com kein Zertifikat erhalten konnten, welches einen Markennamen wie z. B. Twitter im Domainnamen enthält. Daher wurde im allgemeinen Konsens der Name `tmetrics` gewählt. Dieser ließ den zusätzlichen Interpretationsspielraum, dass man das `t` sowohl für Twitter als auch für Trend stehen könne.

Damit war der Einstiegspunkt für die Konfiguration des Webserver gegeben. Nachdem im Rahmen der allgemeinen Projekt Anforderungen schnell klar wurde, dass Java als zentrale Programmiersprache zum Einsatz kommen sollte, wurde ein Server benötigt, welcher Java Programme nativ zur Beantwortung der Browser-Requests ausführen kann. Aufgrund vorhandener Erfahrung mit den Produkten der Apache Software Foundation lag die Wahl eines Tomcat als Application Server sehr nahe. Weitere Alternativen wie Jetty oder der sehr mächtige WebSphere von IBM wurden daher nicht weiter untersucht. Obwohl der Tomcat durchaus in der Lage ist statischen Content wie z. B. HTML, CSS und JavaScript auszuliefern, wurde für diesen Teil auf einen klassischen Apache Webserver gebaut. Nebst dem Grund, dass der Apache bei der Auslieferung statischen Contents im Allgemeinen etwas schneller ist als der Tomcat, war auch hier wieder der im Team

vorherrschende Erfahrungsschatz ein ausschlaggebendes Kriterium. Hier standen sich nur grundsätzlich vorhandene Erfahrungen der Konfiguration eines Tomcat mit tiefer gehenden und langjährigen Erfahrungen im Umgang mit dem Apache gegenüber.

Die Konfiguration entsprechender vHosts für den Domainnamen und Subdomains, der entsprechenden Einstellung für eine bestmögliche Verschlüsselung und die Einrichtung einiger Umleitungen um lediglich den Zugriff mittels HTTPS zu gestatten, stellten daher keine größere Herausforderung dar.

Interessanter war jedoch die Verbindung des Tomcat mit dem Apache. Da der Tomcat als eigenständiger Server auf unabhängigen Ports läuft, wäre es möglich gewesen, relativ einfach mittels des Apache-Proxy-Moduls `mod_proxy` die Anfragen, welche an einen bestimmten externen Pfad gerichtet sind, intern auf den HTTP-Port des Tomcat umzuschreiben. Das Modul `mod_jk` ermöglicht jedoch die Kommunikation über das **Apache JServ Protocol** (AJP) und somit eine deutlich tiefer gehende Kommunikation zwischen Application- und Web-Server. Von einfachen Diagnosemöglichkeiten, wie einer Kontrolle durch anpingen ob der Applicationserver noch läuft, bishin zu Anforderungen an Loadbalancing bietet AJP somit deutliche Vorteile gegenüber dem naiven Ansatz mittels HTTP-Proxy und wurde daher hier eingesetzt. Nachdem der Tomcat erfolgreich mit dem Apache verbunden werden konnte, wurden alle externen Zugriffsmöglichkeiten abgeschaltet, so dass der Tomcat nur noch via AJP über den Apache zu erreichen ist.

Um während des Entwicklungsprozesses stets eine nicht zwingend vollständig fehlerfreie Testumgebung, parallel aber auch immer eine funktionsfähige Produktivversion zur Verfügung zu haben, wurden hierzu die Subdomains `www.` und `dev.` eingerichtet. Hierzu wurde der REST Service unter zwei verschiedenen Pfaden auf den Tomcat deployed. Unterhalb der URL `www.tmetrics.de/rest` stand der REST Service somit nativ zur Verfügung. Hinter der URL `dev.tmetrics.de/rest` verbarg sich jedoch zunächst ein Reverse-Proxy, welcher die Anfragen an `www.tmetrics.de/rest_dev` umgeleitet hat. Der statische Content unterhalb dieser beiden Domains wurde einfach aus zwei verschiedenen Verzeichnissen geliefert.

Die nächste Anforderung bestand darin, die fertig kompilierten REST-Projekte auf den Tomcat zu deployen. Hierzu wird einfach auf die Tomcat eigene Manageroberfläche zurückgegriffen. Damit lässt sich eine `.war` Datei einfach hochladen und auf einen bestimmten Pfad deployen. Dies stellte im Gegensatz zum Einbinden via Dateisystem und anschließendem Reload des Tomcat die einfachere und damit bevorzugte Möglichkeit gar.

Normalerweise ist der Zugang zu der Management-Konsole lediglich über ein Passwort mittels des *Basic Auth* Verfahrens gesichert, was ohne eine sichere Verbindung über HTTPS ein gewisses Sicherheitsrisiko geboten hätte, da das Passwort in jedem Request des Browsers mitgeschickt wird und somit ohne große Mühen von potentiellen Angreifern hätte ausgelesen werden können. Da wir bereits eine gesicherte Verbindung einsetzen und die Management-Konsole wie alle anderen Tomcat-Applikationen über den Apache ausgeliefert werden, bestand an dieser Stelle kein weiterer Handlungsbedarf und die Konfiguration des Tomcat war somit abgeschlossen.

Von den vorhandenen Zertifikaten profitierte ebenfalls die Einrichtung der Mailserver. So konnten sowohl der SMTP-Server postfix als auch der IMAP- und POP-Server dovecot problemlos für verschlüsselte Verbindungen eingerichtet werden. Die Webmail-Anwendung SquirrelMail (<https://mail.tmetrics.de>) nutzt zur Beschleunigung der Zugriffe noch das imapproxy Paket. Somit konnte zur Registrierung der verschiedenen Twitter-Accounts, die für die Minions notwendig waren bequem über bekannte lokale Clients oder ein Webinterface gearbeitet werden.

Als letzter Service sei an dieser Stelle der MySQL-Server und dessen Konfiguration erwähnt. Das reine Installieren des MySQL-Servers ging wie auch bei dem Tomcat und Apache über einen Shell-Aufruf nicht hinaus, und auch die weitere Konfiguration des MySQL-Servers hielt sich zunächst in Grenzen. Aus Sicherheitsgründen wurde der Zugang von außen komplett blockiert, sodass nur von localhost aus auf den MySQL-Server zugegriffen werden konnte. Diese Einstellung wurde im Laufe des Seminars verworfen, da es zu Testzwecken nötig wurde auch von außen Zugriff auf den MySQL-Server zu erhalten.

Da bei der ursprünglichen Konfiguration des MySQL-Servers aber auf die Einrichtung einer Verschlüsselung verzichtet wurde, wurde eine minimale Sicherheit der Datenbank durch unterschiedliche Benutzer umgesetzt. Nicht alle Datenbankbenutzer haben also von außen Zugriff auf die Datenbanken und jene, welche externen Zugriff haben, können nicht alle Daten verändern. Der Verzicht auf eine verschlüsselte Datenbank-Verbindung war von verschiedenen Gründen getrieben. Zum einen sollten unnötige Performance-Einbußen vermieden werden, zum anderen konnte so ein potentieller *point of failure* der Verbindung von Java zur Datenbank konsequent vermieden werden. Zumal nach anfänglicher Planung im späteren Produktivbetrieb sowieso wieder nur lokale Zugriffe stattfinden würden, wurde weiterer Aufwand an dieser Stelle für unnötig erachtet.

Die Grundkonfiguration des Servers war somit gegeben und der Programmierarbeit stand nichts mehr im Wege.

### 4.4.3. Ubuntu-Update auf virtuellem System

Während die meisten Serverkomponenten so mehr oder weniger in ihrer endgültigen Konfiguration vorlagen, ergab sich in Bezug auf den MySQL-Server in den ersten zwei Wochen des Projektes bereits größerer Änderungsbedarf.

Um keine der durch Twitter gelieferten Information zu verlieren war es notwendig, mindestens die Tweets in dem Zeichensatz `utf8mb4_general_ci` zu speichern. Diese 4 Byte lange UTF8-Multibyte-Variante wurde von der eingesetzten MySQL-Version jedoch leider nicht unterstützt. Da auf der gegebenen Ubuntuversion keine neuere MySQL-Version verfügbar war, wurde somit ein Upgrade der Ubuntuversion notwendig. Die Erwartungen, dass das Update eines Ubuntu-Systems so reibungslos funktioniert, wie man es von seinem privaten Computer gewohnt ist, wurden jedoch schnell zerstreut. Nebst einiger konfigurationsbedingter Fehlermeldungen ließen sich einige kernelbezogene Fehlermeldung nicht ausmerzen. Eine daraufhin an den Server4You-Support gerichtete Nachfrage ergab, dass die virtuellen Systeme einen gemeinsamen Kernel einsetzen und dass dieser *shared kernel* eben nicht mit Ubuntu 12.04 LTS kompatibel sei. Man wolle unseren vServer aber auf einen anderen *shared host* umziehen, sodass ein Upgrade rein prinzipiell möglich sein sollte. Da sich vor allem im Bereich der Mailserver-Konfiguration die Struktur der Konfigurationsdateien nahezu vollständig geändert hatte, lief der erste Versuch auch mit neuem *shared kernel* nicht fehlerfrei durch. Die aus den Fehlversuchen gewonnen Erfahrungen trugen jedoch dazu bei, dass der zweite Versuch mit dem neuen Kernel von Erfolg gekrönt war, sodass nach einigem Anpassen diverser Konfigurationsdateien endlich die neue Ubuntuversion fehlerfrei lief und somit auch die neue MySQL-Version inklusive der benötigten Kollation angeboten werden konnte.

Das Update der Ubuntuversion brachte nebst neuen Versionen an nahezu allen Serverkomponenten ebenfalls neue Versionen von Apache und OpenSSL mit sich. Die neuere Version der OpenSSL Bibliothek bot damit die Möglichkeit TLS v1.2 einzusetzen, womit die Sicherheit der HTTPS-Verschlüsselung letztendlich über das durchschnittliche Niveau der deutschen Onlinebanking-Anbieter gebracht werden konnte.

Ein weiterer Vorteil durch die Verwendung einer neueren Apache-Version ergab sich im Zusammenhang mit der Verwendung des von Google zur Verfügung gestellten Apache-Moduls `mod_spdy`. Dies setzt als Mindestanforderung an den Apache die nun zur Verfügung stehende Version voraus.

#### 4.4.4. Performance-Optimierungen

Da sich im Rahmen der Frontend Entwicklung ein Problem mit der durchschnittlichen maximalen Anzahl paralleler Verbindungen, die ein Browser zu einem Webserver offen halten kann, ergab, wurde versucht dieses Problem serverseitig durch die Verwendung des `mod_spdy` zu beheben. Das SPDY-Apache-Modul verspricht durch den Aufbau lediglich einer echten TCP-Verbindung zwischen Browser und Server die Performance durch Reduzierung des mehrfachen TLS-Handshake zu steigern. Dieses Modul dient der Erweiterung des HTTP-Protokolls bzw. der Einführung des SPDY-Protokolls. Das Modul wird von Google implementiert und zur Verfügung gestellt, wobei das SPDY-Protokoll mittlerweile von den meisten gängigen Browsern unterstützt wird. Desweiteren dient das SPDY-Protokoll bzw. einige hier eingesetzte Techniken als Vorlage für die aktuell in den entsprechenden Gremien zur Abstimmung befindliche HTTP 2.0 Protokollversion. Beim Einsatz des HTTP-Protokolls in der üblichen Version 1.1 muss für jede Ressource eine eigene Anfrage an den Server gestellt werden und somit jedes mal eine neue verschlüsselte Verbindung ausgehandelt werden. Das SPDY-Protokoll bündelt diese Anfragen innerhalb einer einzigen gesicherten TCP-Verbindung. Zunächst schien dies die Lösung zu sein, mehrere "logische" und vor allem parallele Anfragen als gewohnt anbieten zu können. Diese Annahme stellte sich jedoch als Fehler heraus. Tests ergaben, dass auch mit `mod_spdy` nicht mehr parallel logische Verbindungen möglich waren als zuvor. Somit war für unser System an dieser Stelle lediglich in Bezug auf die Auslieferung des statischen Content ein Performance-Vorteil vorhanden. Wir haben uns dennoch dafür entschieden, weiterhin `mod_spdy` zu verwenden. Browser die dieses Protokoll nicht unterstützen fallen transparent auf HTTP(S) 1.1 (bzw. 1.0) zurück.

Aufgrund von Problemen mit der parallelen Verarbeitung von PHP-Skripten im Zusammenhang mit `mod_spdy` lief das eingesetzte `phpMyAdmin` nicht mehr, sodass für das PHP-Parsing noch ein `fgci`-Modul eingesetzt wurde. Nach der Installation der relevanten Apache-Module `libapache2-mod-fcgid` und `php5-cgi` und dem setzen des `fgci`-Handlers für `.php`-Dateien war aber auch dieses Problem behoben.

Da gegen Ende des Seminars die Performance von Datenbankoperationen nach und nach abnahm, nutzten wir einen kostenlosen Probemonat eines leistungsstärkeren dedizierten Servers. Eine Verbesserung bei der Datenbank und beim Daemon konnte festgestellt werden. Leider beseitigte das dennoch nicht alle Performance-Probleme, sodass das Team an

dieser Stelle feststellte, dass eine Überarbeitung des Datenbankschemas wohl viel versprechender wäre als eine reine Leistungssteigerung der Hardware.

Um während der Präsentation dennoch ein einigermaßen performantes System zur Verfügung zu haben, wurden sämtliche Anfragen an den REST-Service mittels des Apache-Proxy-Moduls zwischengespeichert. Hierzu wurde der Aufruf des Rest Services auf einen anderen Pfad umgelegt (`www.tmetrics.de/rest_prod`) und auf dem alten Pfad (`www.tmetrics.de/rest`) ein Reverse-Proxy eingesetzt. Nach einigen Untersuchungen in Bezug auf die verschiedenen Caching-Möglichkeiten, stellte sich heraus, dass leider nicht alles wie gewünscht gecached werden kann, da in den Anfrage URLs Query-Parameter wie z. B. `id=1` verwendet werden. Daher musste am REST-Service eine kleine Änderung vorgenommen werden, sodass alle Antworten nun mit einem Expires-Header ausgesendet werden. Dies hat zur Folge, dass alle Antworten des REST-Service sowohl im Browser als auch auf dem Proxy für 90 Minuten zwischengespeichert werden. Im Zusammenhang mit dem Proxy ergibt sich also die Situation, dass sobald ein Besucher sich einmal die Auswertung für z. B. Merkel hat anzeigen lassen, diese direkt und unmittelbar auch für alle anderen Besucher zur Verfügung steht. Somit konnte eine reibungslose Demonstration gewährleistet werden.



## 5. | Komponenten

### 5.1. Sentiment

Eines der Ziele von TMetrics ist es, ein umfassendes und detailliertes Meinungsbild zu einem Thema zu erstellen. Diesem Zweck dient die Sentimentanalyse: die „rechnerische Handhabung von Meinungen, Sentiment und Subjektivität in Text“ [34]. Unter Sentiment wird dabei ein Gefühl verstanden, das sich auf ein bestimmtes Ziel richtet, beispielsweise Ablehnung oder Zustimmung zu einer politischen Meinung.

Die Datenmengen, die in modernen sozialen Medien anfallen und die auch als „Big Data“ verschlagwortet worden sind, machen eine manuelle Analyse impraktikabel. Sentimentanalyse auf Twitterkommunikation anzuwenden heißt also, die themenspezifischen Meinungen einer Gruppe von Menschen automatisiert zu analysieren und zu quantifizieren. Dabei wird die Annahme zugrundegelegt, dass sich der emotionale Gehalt eines Tweets in einer trinären Klassenzuordnung (gut, neutral, negativ) oder einer Zahl von -1 bis +1, im Folgenden Sentimentwert genannt, ausdrücken lässt.

Im Unterabschnitt 5.1.1 werden eine Reihe von theoretischen Modellen vorgestellt, um den Sentimentwert eines Texts zu bestimmen. Im Unterabschnitt 5.1.2 wird das Java-Paket beschrieben, in dem diese Modelle implementiert wurden. Im Unterabschnitt 5.1.2 wird schließlich beschrieben, wie die Berechnung der Sentimentwerte in diesem Paket in die restliche Architektur mit Daemon und REST-Service eingebunden ist und wie die Sentimentwerte schließlich im Front-End angezeigt werden.

#### 5.1.1. Theoretische Grundlagen

Eine simple Methode, den Sentimentgehalt eines Tweets näherungsweise zu bestimmen, wird von Twitter selbst verwendet. Es werden eine Liste von positiven Emoticons und eine Liste von negativen Emoticons erstellt. Wenn ein positives Emoticon im Tweet enthalten ist, wird der Tweet als positiv aufgefasst; ist ein negatives Emoticon enthalten, wird er

*Tabelle 5.1.: Trainingsdaten*

Text	Sentiment
I love puppies	+1
I hate puppies	-1

als negativ verstanden. Im Beispielsatz „I love puppies :-)” wird beispielsweise das „:-)” als positiv erkannt. Diese Methode ist einfach und nach unserer Erfahrung in der Hinsicht einigermaßen verlässlich, dass tatsächlich die meisten so identifizierten Tweets positiv sind. Allerdings ist sie ungeeignet, um ein detailliertes Meinungsbild zu erstellen, weil nur ein Bruchteil aller Tweets überhaupt ein Emoticon enthält.

Statt Listen von positiven und negativen Emoticons lassen sich Listen von Wörtern verwenden. Wörterbücher dieser Art sind aufwändig zu erstellen, da sie oftmals tausende von Wörtern enthalten, aber einige solcher Listen sind frei im Internet verfügbar. Im Satz „I love puppies :-)” wird mithilfe eines Wörterbuchs beispielsweise das „love“ als positiv erkannt. Die Bewertungen der Einzelwörter werden dann aggregiert, um eine Bewertung des Textes zu berechnen.

Ein naiver Wörterbuchansatz allein ist allerdings ebenfalls ungeeignet, den Sentimentwert eines Tweets zu bestimmen. Zum Beispiel würde ein Wörterbuch in der Kombination „not good“ ein positives Wort sehen. Auch Redewendungen und Ironie werden so nicht erkannt. Daher werden teils zusätzlich Listen von Phrasen verwendet, oder die Wörterbücher werden gemeinsam mit Regelwerken verwendet, um z. B. den Sentimentwert eines Wortes umzukehren, das auf „not“ folgt. Auch solche Regelwerke müssen von Menschen mühevoll erstellt werden. Angesichts der Komplexität natürlicher Sprachen verwundert nicht, dass sie fehleranfällig und unvollständig sind.

Eine Forschungsdisziplin, die Abhilfe verspricht, ist die des maschinellen Lernens (*machine learning*). Im Gegensatz zu Emoticon- und Wörterbuch-Listen und Regelwerken müssen Zusammenhänge hier nicht mehr von Menschen explizit formuliert werden, um dann von der Maschine für deduktive Schlussfolgerungen verwendet zu werden. Stattdessen lernt die Maschine induktiv anhand von Beispielen. Das so gewonnene „Wissen“ erlaubt nach der Trainingsphase die Klassifikation neuer Daten anhand von Beispielen. Die Erstellung dieser Beispiele, Trainingsdaten genannt, ist wesentlich leichter als die Erstellung komplexer Regelwerke. Tabelle 5.1 zeigt zwei Beispielsätze, die als Trainingsdaten dienen können.

Tabelle 5.2.: Trainingsdaten als Merkmalsmatrix

I	love	hate	puppies	Sentiment
1	1	0	1	+1
1	0	1	1	-1

Im Gebiet des maschinellen Lernens werden eine Reihe von Algorithmen angewandt. Dazu zählen zum Beispiel lineare und logistische Regression, *Support Vector Machines* und neuronale Netzwerke. Im Rahmen dieses Projekts fiel die Entscheidung auf die Implementierung der linearen Regression. Die Wahl eines vergleichsweise einfachen Algorithmus erlaubte es uns, diesen vollständig und korrekt selbst zu implementieren, anstatt auf eine vorhandene Implementierung in Form einer Bibliothek zurückgreifen zu müssen. Dies war im akademischen Kontext mit der Zielsetzung eines möglichst großen Lerneffekts erwünscht.

Der linearen Regression liegt folgendes Modell zugrunde:

$$y = h_{\theta}(x) + \epsilon = \theta_0 x_0 + \theta_1 x_1 + \dots + \theta_n x_n + \epsilon$$

Für jeden Trainingsdatensatz ist  $y$  das tatsächliche Sentiment des Texts, das anhand der Merkmale  $x_1 \dots x_n$  vorhergesagt werden soll ( $x_0 = 1$ ). Die Werte der Parameter  $\theta_0, \dots, \theta_n$  sind unbekannt. Der Fehlerterm  $\epsilon$  gibt die Abweichung der Schätzung anhand der Merkmale und Parameter vom tatsächlichen Wert an. Betrachtet man alle Trainingsdaten gleichzeitig, kommt die vektorisierte Form der Gleichung zum Einsatz:

$$y = h_{\theta}(x) + \epsilon = X \cdot \theta + \epsilon$$

$y$  und  $\epsilon$  sind jetzt Vektoren.  $X$  ist eine Matrix, die Merkmals- oder Featurematrix, die in den  $m$  Zeilen die einzelnen Trainingsdaten enthält. Die  $n + 1$  Spalten sind die einzelnen Merkmale. Die Einträge der Matrix geben an, ob und in welcher Quantität ein Merkmal in einem Trainingsdatensatz vorliegt. Tabelle 5.2 zeigt die Merkmalsmatrix.

Die Aufgabe der linearen Regression besteht nun darin, die Einträge im Fehlervektor  $\epsilon$  zu minimieren. Dies geschieht typischerweise mit der Methode der kleinsten Quadrate, d. h. die Summe der Quadrate der Fehlerterme soll minimal werden:

$$\min \frac{1}{2m} \sum_{i=1}^m (h_{\theta}(x^{(i)}) - y^{(i)})^2$$

Für die Suche nach dem Minimum dieser Funktion existiert eine geschlossene Lösung, die jedoch eine Matrixinversion einer  $n \times n$ -Matrix erfordert mit  $n$  als Anzahl der Merkmale. Da Matrixinversionen typischerweise eine Laufzeit in  $\mathcal{O}(n^3)$  hat und  $n$  schon bei der Verwendung von Unigrammen (Wörtern) als Merkmalen schon bei einigen hundert Tweets eine Zahl im Tausenderbereich erreicht, ist dieser Algorithmus für die Sentimentanalyse ungeeignet.

Eine Alternative ist das Gradientenabstiegsverfahren, bei dem zu Beginn beliebige Parameterwerte gewählt werden (z.B. 0 für jeden Parameter) und das in jeder Iteration einen kleinen Schritt in Richtung des Gradienten macht, d.h. die Parameterwerte bei jeder Iteration so verändert, dass der Funktionswert geringer wird. Diese Methode läuft zwar Gefahr, zu einem lokalen Optimum zu konvergieren, ohne das globale Optimum zu finden. Da aber die Kostenfunktion konvex ist, d.h. keine lokalen Minima außer dem globalen Minimum hat, kann man diese Schwäche hier vernachlässigen. Im Gradientenabstiegsverfahren werden in jeder Iteration  $i$  die Sentimentwerte mit den aktuellen Parameterwerten geschätzt und die neuen Parameterwerte wie folgt berechnet:

---

**Algorithmus 1** Gradientenabstiegsverfahren für die lineare Regression

---

**Eingabe:**  $X \in \mathbf{R}^{m \times n+1}$  Merkmalsmatrix mit den Einträgen  $x_j^i$ ,  $\epsilon$  Konvergenzmaß,  $\alpha$  Schrittweite

**Ausgabe:**  $\theta \in \mathbf{R}^{n+1}$  Parametervektor mit den Einträgen  $\theta_j$

```

1: for ( $j = 1; j < n + 1; j++$ ) do
2:    $\theta_j = 0$ 
3: end for
4:  $e_{current} = \inf$ 
5: repeat
6:    $h_\theta = X \cdot \theta$ 
7:    $e_{last} = e_{current}$ 
8:    $e_{current} = \frac{1}{2m} \sum_{i=1}^m (h_\theta(x^{(i)}) - y^{(i)})^2$ 
9:   for ( $j = 1; j < n + 1; j++$ ) do
10:     $\theta_j = \theta_j - \alpha \frac{1}{m} \sum_{i=1}^m (h_\theta(x^{(i)}) - y^{(i)}) x_j^{(i)}$ 
11:   end for
12: until  $e_{last} - e_{current} \leq \epsilon$ 

```

---

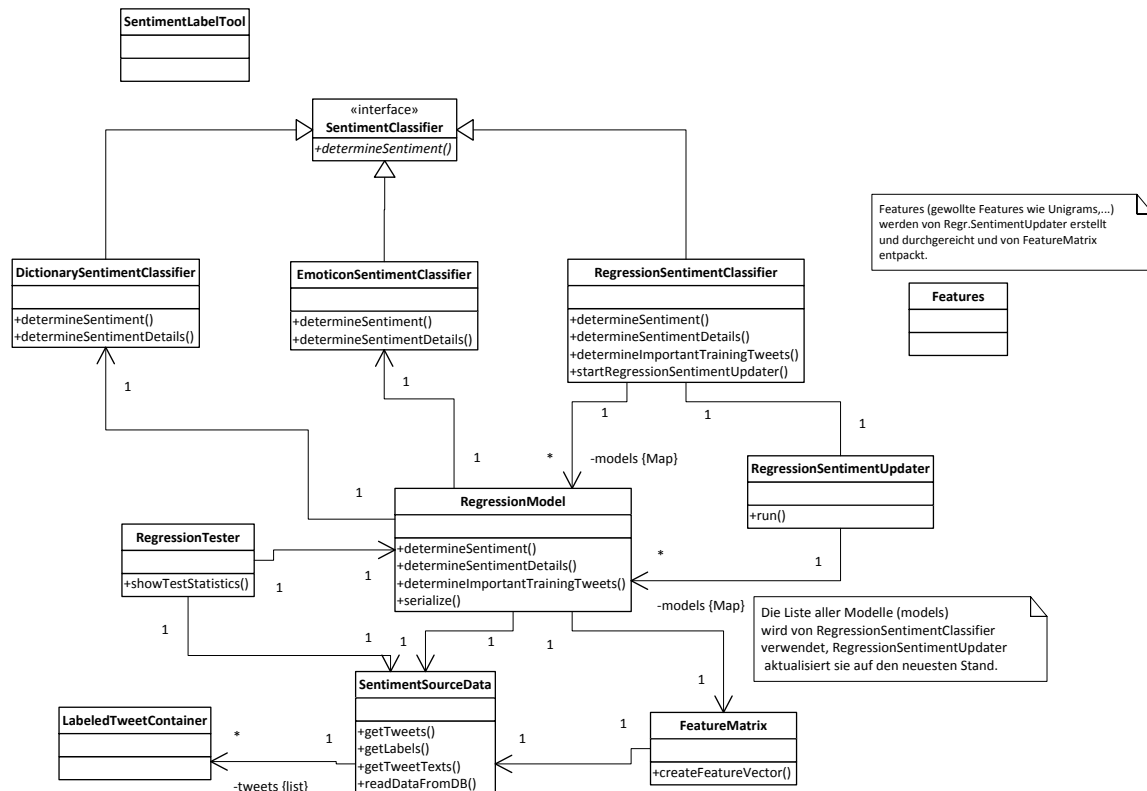


Abbildung 5.1.: Architektur des Pakets `daemon.sentiment`

### 5.1.2. Implementierte Klassifikatoren

Abbildung 5.1 zeigt die Architektur des Java-Pakets `daemon.sentiment`. Es wurden nur die relevanten öffentlichen Methoden gekennzeichnet, um die Grafik übersichtlich zu halten.

Das Interface `SentimentClassifier` definiert die Methode `determineSentiment()`. Diese berechnet zu einem gegebenen Text in einer gegebenen Sprache den Sentimentwert. Drei Klassifikatoren implementieren dieses Interface, wobei jede der Implementierungen eine der vorgestellten Methoden umsetzt, Sentiment zu berechnen. Diese Klassen bilden die Schnittstelle des Sentiment-Pakets nach außen: Sie können von Daemon oder REST-Service instanziiert werden (im Endprodukt wird der akkurateste Classifier, der `RegressionSentimentClassifier`, instanziiert). Der Wörterbuch- und der Emoticon-Klassifikator erledigen alle anfallenden Aufgaben selbst. Der wichtigste Klassifikator, `Re-`

`gressionSentimentClassifier`, greift zur Bewältigung seiner Aufgaben auf eine Reihe weiterer Klassen zurück, in deren Zentrum die Klasse `RegressionModel` steht.

## Die Schnittstelle nach außen: Implementierungen von `SentimentClassifier`

Der `EmoticonSentimentClassifier` berechnet den Sentimentwert eines Textes auf Grundlage einer Emoticon-Liste, die in einem Abgleich mit dem Ergebnis der Twitter-Suche entstanden ist. Mit einer Twitter-Suche nach *Merkel :-)* werden Tweets ausgegeben, die Twitter als positiv erkennt und die beispielsweise auch die Emoticons *:-D* oder *:-P* enthalten; analoges gilt für negative Tweets.

Der `DictionarySentimentClassifier` verwendet ein Wörterbuch, um den Sentimentwert eines Texts zu berechnen. Dabei kommen das Wörterbuch von Liu [22] und die folgende Aggregierungsfunktion:

$$\frac{\#pos - \#neg}{\#pos + \#neg}$$

Das Vorgehen ist damit dasselbe wie bei Stieglitz und Dang-Xuan (2012) [36], allerdings mit einem anderen Wörterbuch, da das Wörterbuch von Liu im Gegensatz zum Wörterbuch des dort verwendeten Tools LIWC frei verfügbar ist. Beide beinhalten keine Gewichtungen für einzelne Wörter (z. B. ist „super“ genauso positiv wie „good“), sodass die Aggregationsfunktion die Häufigkeit von positiven bzw. negativen Wörtern verwendet.

Das Wörterbuch wird „roh“ verwendet, ohne grammatikalische Regeln, trotz der Warnung von Liu, dies nicht zu tun, sodass z. B. „not happy“ positiv ist. Da aber vorgesehen ist, das Wörterbuch durch einen Machine-Learning-Algorithmus zu ersetzen, schien eine dahingehende Erweiterung der Wörterbuchklasse unsinnig.

Das Wörterbuch bewertet nur englischsprachige Tweets. Um zu erkennen, ob ein Tweet englischsprachig ist, wird der Rückgabewert der Twitter API verwendet, die zu jedem Tweet einen zweistelligen ISO-Sprachcode liefert. Diese Klassifikation hat sich als ausreichend genau herausgestellt. Nicht-englischsprachige Tweets erhalten einen Sentimentwert von `null`.

Die Klasse `RegressionSentimentClassifier` implementiert das Interface `SentimentClassifier` und verwaltet die verschiedenen Regressionsmodelle. In einer Hashtabelle liegt für jede Sprache, für die genügend Trainings-Tweets vorliegen, ein Regressionsmodell. Als Schlüssel werden wiederum die zweistelligen ISO-Sprachcodes verwendet. Beim Start versucht der Classifier zunächst, bestehende, serialisierte Modelle von der Festplatte zu

lesen und diese zu verwenden. Mit der Methode `startRegressionSentimentUpdater()`, die vom Daemon aufgerufen wird, wird ein neuer Thread gestartet, der `RegressionSentimentUpdater` (siehe 5.1.3), der sich darum kümmert, die Modelle neu zu trainieren, falls sie nicht auf der Festplatte lagen, und aktuell zu halten, wenn neue Trainingsdaten vorliegen. Schließlich stellt auch dieser Classifier die Methode `determineSentiment()` zur Verfügung, die basierend auf der übergebenen Tweet-Sprache das richtige Modell auswählt (falls verfügbar) und den berechneten Sentimentwert des Tweets zurückgibt.

## Der Unterbau: RegressionModel und seine Helfer

Die Klasse `RegressionSentimentClassifier` greift für die Berechnung auf weitere Klassen zurück, in deren Mittelpunkt das `RegressionModel` steht.

`RegressionModel` kapselt ein Modell, das bei Instanziierung mit einem spezifischen Satz Trainingsdaten trainiert wird, d. h. die Merkmalsmatrix bestimmt und darauf basierend die Parameter schätzt. Für die anderen Klassen im Paket (insbesondere `RegressionTester`) stellt es Methoden zur Verfügung, die bestimmte Statistiken (z. B. die verwendeten Trainingsdaten, die geschätzten Parameter) ausgeben oder in TSV-Dateien exportieren, aber die einzigen `public`-Methoden sind auch hier `determineSentiment()` und dessen Variationen, die das Sentiment eines neuen Textes auf Basis der Parameter bestimmen. Die Methode `trainModelGradientDescent()` implementiert den eigentlichen Algorithmus (siehe 1). Der Algorithmus wird mit Nullen als Parameterwerte initialisiert und so häufig wiederholt, bis sich die Schätzungen nur noch unwesentlich verändern.

`FeatureMatrix` stellt die Matrix dar, die die extrahierten Features bzw. Einflussfaktoren enthält. Es kommen die folgenden Features zum Einsatz, welche am Beispielsatz „I love puppies but I don’t like kittens :-“ veranschaulicht werden.

- **Anzahl positiver Emoticons:** Ein Rückgabewert aus `EmoticonSentimentClassifier`. Im Beispielsatz 1.
- **Anzahl negativer Emoticons:** Ein Rückgabewert aus `EmoticonSentimentClassifier`. Im Beispielsatz 0.
- **Emoticon-Sentiment:** Die Sentiment-Schätzung aus `EmoticonSentimentClassifier`. Im Beispielsatz 1.
- **Anzahl positiver Wörter:** Ein Rückgabewert aus `DictionarySentimentClassifier`. Im Beispielsatz 2.

- **Anzahl negativer Wörter:** Ein Rückgabewert aus `DictionarySentimentClassifier` Im Beispielsatz 0.
- **Dictionary-Sentiment:** Die Sentiment-Schätzung aus `DictionarySentimentClassifier`. Im Beispielsatz 1.
- **Unigrams:** Einzelne Wörter, z. B. „I“, „love“, ...
- **Bigrams:** Zwei aufeinanderfolgende Wörter, z. B. „I love“, „love puppies“, ...
- **Trigrams:** Drei aufeinanderfolgende Wörter, z. B. „I love puppies“, „love puppies but“, ...
- **Fourgrams:** Vier aufeinanderfolgende Wörter, z. B. „I love puppies but“, „love puppies but I“, ...

Darüber hinaus gibt es zwei Optionen, die die bestehenden Features modifizieren, wenn sie angeschaltet sind:

- **Negation:** Jedem Wort, das als negiert erkannt wird, wird ein Suffix „\$NEG\$“ angehängt. Aus dem Unigram „like“ wird z. B. „like \$NEG\$“
- **Part-of-Speech-Tagger:** Jedem Wort wird seine Wortart in Form eines *part of speech tag* als Suffix angehängt. Aus dem Unigram „like“ wird z. B. „like \$VB\$“ (für *verb*)

`SentimentSourceData` spiegelt die Regressionsdaten wieder, kapselt also eine Menge von Tweets. Dabei handelt es sich entweder um die Trainings- oder die Testdaten. Die Klasse stellt Methoden bereit, um diese Daten aus einer Datenbank zu laden. `Labeled-TweetContainer` kapselt einen einzelnen Tweet und das dazugehörige, von Menschen bestimmte, echte Sentiment (*Label*).

`RegressionTester` kapselt den Test eines bestimmten Modells mit einem bestimmten Satz Testdaten. In der produktiven Nutzung sollen natürlich alle verfügbaren Daten als Trainingsdaten verwendet werden, um eine möglichst gute Vorhersagekraft zu erreichen. Die Tester mit ihrer Main-Methode lassen sich aber in der Entwicklung und Analyse nutzen, um nur einen Teil der Daten als Trainingsdaten zu verwenden. Ebenso können mehrere Tester für dasselbe Modell erstellt werden, um die Vorhersagekraft des Modells an unterschiedlichen Testdatensätzen zu evaluieren, oder Tester für mehrere Modelle, um die Modelle zu vergleichen.

Außerdem wurden im Paket `tmetrics.util` Helfer implementiert:



`SparseMatrix` ist eine Matrix. Einfach ein zweidimensionales Array zu nutzen, wäre bei Matrizen in dieser Größenordnung (schon bei 123 gelabelten Tweets gab es 814 Spalten) eine Speicherplatzverschwendung, weil sehr viele Nullen gespeichert werden. Daher werden hier nur die Positionen gespeichert, die ungleich 0 sind. Die Matrix ist als Liste von Listen von Tupeln realisiert. Jede innere Liste ist eine Zeile. Jedes Tupel ist (Spalte, Wert). Der Grund für diese Entscheidung ist, dass die Matrix typischerweise zeilenweise geschrieben und gelesen wird, Operationen, die so in  $\mathcal{O}(1)$  möglich sind. Zudem ist die Anzahl der Spalten von Interesse, weshalb diese in einer Membervariable vorgehalten und bei Änderungen aktualisiert wird. Die Klasse stellt außerdem Methoden für benötigte Operationen (z. B. die Multiplikation der transponierten Matrix mit einem Vektor) zur Verfügung.

`ListUtil` stellt Methoden zur Verfügung, die Berechnungen mit Listen von Floats ermöglichen, welche wir als Vektoren benutzen. Ein Beispiel ist die Skalarmultiplikation zweier Vektoren. `LinearRegression` stellt die eigentliche lineare Regression dar, die als Eingabe eine `FeatureMatrix` und die Labels erwartet. Daraufhin wird eine Regression mittels Gradientenverfahren durchgeführt. Die geschätzten Parameter sowie Werte zur internen Beurteilung (z. B. die Fehler oder geschätzten Werte der Trainingsdaten) können anschließend abgefragt werden.

### 5.1.3. Einbindung in die Architektur

Der Nutzer sieht Sentimentwerte im Front-End in verschiedenen Detailstufen, die in Abschnitt 5.4.5 näher erläutert werden. In der Tweet-Ansicht werden die für einzelne Tweets ermittelten Sentimentwerte direkt aus der Datenbank geladen. Wenn sich der Nutzer die Ergebnisse der Sentiment-Berechnung in detaillierter Form ausgeben lässt, werden die Sentiment-Modelle dagegen von der Festplatte geladen. Wie die Ausgabe dieser detaillierten Informationen umgesetzt ist, zeigt Abbildung 5.2.

Der Nutzer interagiert mit der Anwendung über das im Browser angezeigte Front-End. Seine Anfragen führen (via REST-Service) dazu, dass auf die Modelle zugegriffen wird, die auf der Festplatte liegen. Doch zeitgleich kann der Entwickler Tweets `labeln`, d. h., ihren Sentimentwert manuell bestimmen und so Trainingsdaten schaffen, um die Genauigkeit der Schätzungen zu verbessern. Deshalb müssen die Modelle aktuell gehalten werden, wenn neue Trainingsdaten vorliegen.

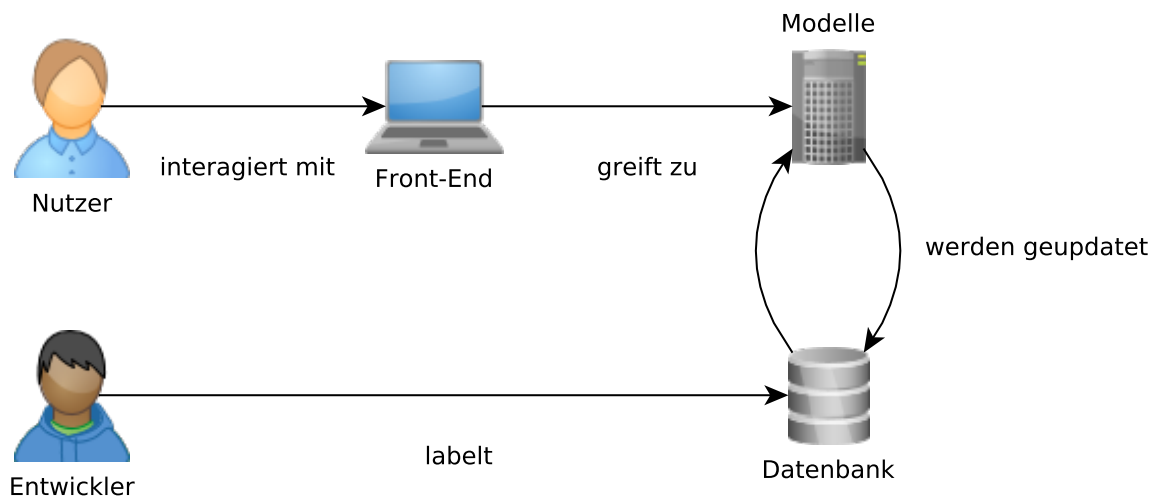


Abbildung 5.2.: Architektur aus Sicht des Sentiment-Pakets

**SentimentLabelTool** ist ein von der restlichen Anwendung unabhängiges Tool, das es ermöglicht, Tweets ohne direkten Zugriff auf die Datenbank und ohne Tools wie PHP-MyAdmin zu labeln. Der Nutzer wählt die Sprache aus, in der Tweets gelabelt werden sollen. Dann wird ein Batch von 1000 Tweets geladen und der erste angezeigt. Der Nutzer gibt seine Bewertung in den Schritten -1, -0,5, 0, 0.5 oder 1 ein und bekommt anschließend den nächsten Tweet angezeigt. Tweets können auch übersprungen werden. Sobald zehn Tweets bearbeitet worden sind, wird der Batch vom Tool an die Datenbank gesendet und der nächste geholt. Der Grund für die unterschiedlichen Batch-Größen liegt darin, dass beim Laden 1000 zufällige Tweets ausgewählt werden. Diese Operation nimmt in MySQL einen sehr langen Zeitraum in Anspruch, dessen Dauer nicht von der Anzahl der abgefragten Tweets abhängt. Das Speichern von Sentimentwerten ist dagegen eine vergleichsweise schnelle Operation.

**RegressionSentimentUpdater** läuft in einem eigenen Thread und kümmert sich um die Aktualisierung der Regressionsmodelle und die Sentiment-Aktualisierung aller Tweets in der Datenbank. Dazu überprüft er zunächst, ob neue Trainingsdaten zu einer beliebigen Sprache in der Datenbank vorhanden sind. Dies ist immer dann der Fall, wenn neue Tweets gelabelt worden sind. In diesem Fall wird eine Regression durchgeführt und das Modell gespeichert. Das Modell steht sofort als Java-Objekt dem **RegressionSentimentClassifier** zur Verfügung und wird für den Zugriff auch nach einem eventuellen Neustart des Daemons sowie für den REST-Service serialisiert und auf der Festplatte gespeichert.

Anschließend wird das Sentiment aller Tweets in der Datenbank mit Hilfe des neuen Regressionsmodells berechnet und aktualisiert. Diese Überprüfung und die erneute Regression werden für alle Sprachen alle 24 Stunden durchgeführt. Das Programm ist somit selbst dafür zuständig, zu erkennen, ob es neue Modelle erstellen kann und es ist außer dem Labeln keine andere manuelle Intervention notwendig, um bessere Sentimentschätzungen zu erhalten.

#### 5.1.4. Diskussion

Es lassen sich zwei Herausforderungen identifizieren, die sich bei der Umsetzung der Sentiment-Vorhersagen ergaben, nämlich die Überanpassung (*overfitting*) und die Datenbankperformance.

Voraussetzung für eine erfolgreiche Sentimentanalyse ist, dass die Sentimentwerte zu vielen Texten bereits bekannt sind. Die lineare Regression findet die optimalen Gewichte für die Vorhersage der Labels zu den Trainingsdaten. Dies bedeutet nicht notwendigerweise, dass neue Daten auch gut vorhergesagt werden. Der Begriff der Überanpassung an die Trainingsdaten beschreibt das Phänomen, dass Eigenheiten der Trainingsdaten, die nicht für andere Daten verallgemeinert werden können, vom Klassifikator zur Bewertung herangezogen werden. Überanpassung tritt auf, wenn die Zahl der Parameter bzw. Features zu hoch ist für die Anzahl der Datensätze.

Das heißt, dass die Überanpassung zu verringern ist, indem die Zahl der Datensätze erhöht oder die Zahl der Parameter gesenkt wird. Je mehr Trainingsdaten vorliegen, die zufällig aus dem Korpus ausgewählt sind, desto geringer wird der Einfluss einzelner Zufälle, aber Überanpassung kann nie ganz ausgeschlossen werden. Zum Beispiel könnte ein Begriff wie Irak auch über Jahre hinweg mit schlechten Nachrichten und erhitzten Debatten assoziiert sein. Mit diesen Daten trainiert, würde ein Klassifikator ihn (richtigerweise) zur Vorhersage negativer Emotion verwenden, und könnte vermutlich nicht verwendet werden, um Nachrichten über Irak aus einem anderen Themengebiet zu verwenden.

Die Überanpassung hat sich in diesem Projekt wie folgt geäußert: Zur Einschätzung des Sentimentwertes eines neuen Tweets können offensichtlich nur Features verwendet werden, die dem Modell bereits aus den Trainingsdaten bekannt sind. Obwohl mehr als 1000 manuell annotierte Tweets (deutsche und englische) als Trainingsdaten verwendet wurden, kam es regelmäßig vor, dass in den Testdaten Wörter vorkamen, die zweifellos zur emotionalen Aussage des Tweets beitrugen, aber nicht in den Trainingsdaten vorhanden gewesen

waren und somit nicht zur Schätzung des Sentiments verwendet werden konnten. Das Feature wurde nicht erkannt, der entsprechende Parameterwert fehlt im Modell. Da bei einer linearen Regression mit n-Gramm-Features der Sentimentwert des Texts die Summe der Parameter der einzelnen n-Gramme ist, führt dieses Fehlen von Parameterwerten zu einer (absolut) zu niedrigen Schätzung.

Hinzu kommt, dass für den Algorithmus bei einem Mangel an Trainingsdaten nicht ersichtlich ist, welche Features für die Einschätzung des Sentimentwerts wichtig sind und welche nicht. Implizit werden häufige Features stärker gewichtet, denn wenn in einer Iteration des Gradientenabstiegsverfahrens der Parameterwert eines häufigen Features besser wird, hat dies einen höheren Einfluss auf den Fehler des Modells als bei einem seltenen Feature. Dieser Unterschied ist allerdings nur dann ausgeprägt, wenn ausreichend Trainingsdaten vorliegen. Wenn er nicht ausgeprägt ist, wie in unserem Fall, erhalten auch seltene Features, beispielsweise Fourgrams, vergleichsweise hohe Parameterwerte, und häufige Features vergleichsweise niedrige. Wird dieses Modell auf neue Daten angewandt, und werden dabei nur einige der vorliegenden Features erkannt, kommt es hierdurch wiederum verstärkt zu einer (absolut) zu niedrigen Schätzung. Es liegt ein Bias zugunsten absolut niedriger Sentimentwerte vor.

Zur Behebung des Problems wurden unterschiedliche Lösungsansätze versucht. Der vielversprechendste war, die Features auf all jene zu begrenzen, die mindestens in zwei Trainingsdatensätzen vorkamen. So nahmen wir zwar in Kauf, dass noch weniger Features wiedererkannt würden, erhofften uns aber auch, dass bei den tatsächlich wiedererkannten Features die Parameterwerte absolut höher sein würden. Dennoch ließ sich der Bias so nicht beseitigen. Schließlich beschlossen wir, das Problem zu umgehen: Vorrangiges Ziel war die Einordnung der Tweets in negative, neutrale und positive. Diese Einordnung funktioniert trotz des Bias gut, wenn die Grenzen zwischen den Kategorien ebenfalls auf absolut niedrige Werte gesetzt werden.

Im Ergebnis empfehlen wir für den praktischen Einsatz die weitere Erhöhung der Zahl der Trainingstweets und möglicherweise die Verringerung der Anzahl Features mithilfe von Algorithmen zur *feature selection*. Um den beschriebenen Bias zu verhindern, erscheinen nichtlineare Algorithmen vielversprechend, die besser mit dem Fehlen von Features umgehen können.

Neben der Überanpassung war die Datenbank-Performance eine Schwierigkeit. Gerade der `RegressionSentimentUpdater` hat die komplexe Aufgabe, die Sentimentwerte aller Tweets in der Datenbank zu ändern, wenn neue Daten gelabelt worden sind. Im pro-

duktiven Einsatz kommt dies zwar kaum vor, während der Entwicklung dagegen umso häufiger. Bei mehreren Millionen Tweets nehmen die nötigen Datenbankoperationen eine große Menge Zeit und Ressourcen in Anspruch. Diese Aufgabe ist erfolgreich gelöst worden. Als hilfreich hat sich insbesondere das Profiling erwiesen, also das Aufzeichnen von Methodenlaufzeiten und verwendetem Speicher, um Bottlenecks zu identifizieren. Eine akzeptable Laufzeit und ein geringer Speicherverbrauch wurden schließlich durch die Verwendung geeigneter *batched statements* in Verbindung mit manuellen Commits sowie einem *streaming result set*.

## 5.2. Cluster-Analyse

In der frühen Phase des Projektseminars wurden verschiedene Ideen für das Projekt vorgeschlagen. Eine der Ideen beschäftigte sich mit dem Gedanken, die Tweets als Punkte zu repräsentieren und damit explorativ zu arbeiten. Dabei sollte es möglich sein, durch die Entfernung zwischen den Tweets zu sehen, ob zwei Tweets zu einem Thema gehören und eher die selbe Meinung vertreten oder nicht. Die Abbildung 5.3 illustriert die vorgestellte Idee, wobei die einzelnen Symbole die Tweets repräsentieren und zu Clustern zusammengefasst sind. Damit kann ein Benutzer sehen, welche Gruppen sich bilden und wie die Meinungen zu einem bestimmten Thema aussehen. Es könnte dabei um die Entscheidung gehen, welchen Film man sich angucken möchte, aber auch um die Entwicklung eines neuen Produktes, wobei das neue Produkt möglichst die Wünsche der Benutzer erfüllen sollte. Dies motivierte die Implementierung einer Cluster-Analyse für Tweets.

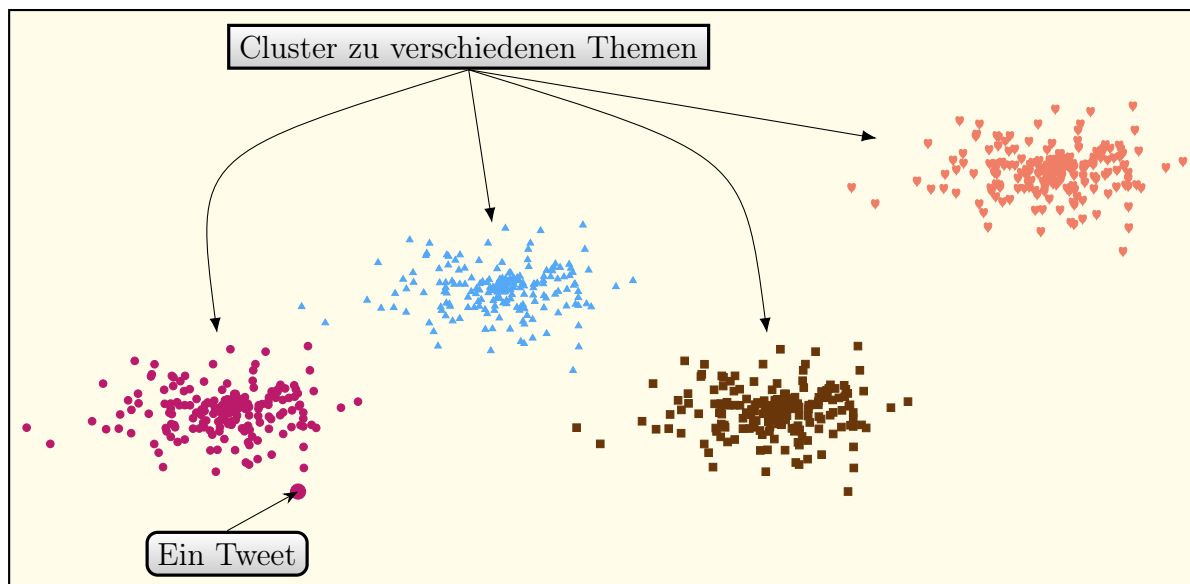


Abbildung 5.3.: Illustration der Idee zur Clusteranalyse von Tweets.

Der Aufbau dieses Kapitels orientiert sich dabei an der grundlegenden Vorgehensweise der Cluster-Analyse, die in Abbildung 5.4 dargestellt ist. Nach Abruf der Daten erfolgt zunächst die Extraktion der Merkmale der zu gruppierenden Objekte. Diese Objekte können sowohl Tweets als auch Hashtags sein.

Bei der Analyse von Tweets stellt die beschränkte Textlänge ein Problem dar, wodurch sich nur wenige Informationen extrahieren lassen. Daher wurden schrittweise verschiedene Ansätze zur Merkmalsextraktion implementiert, zum Beispiel die Erfassung der Hashtags

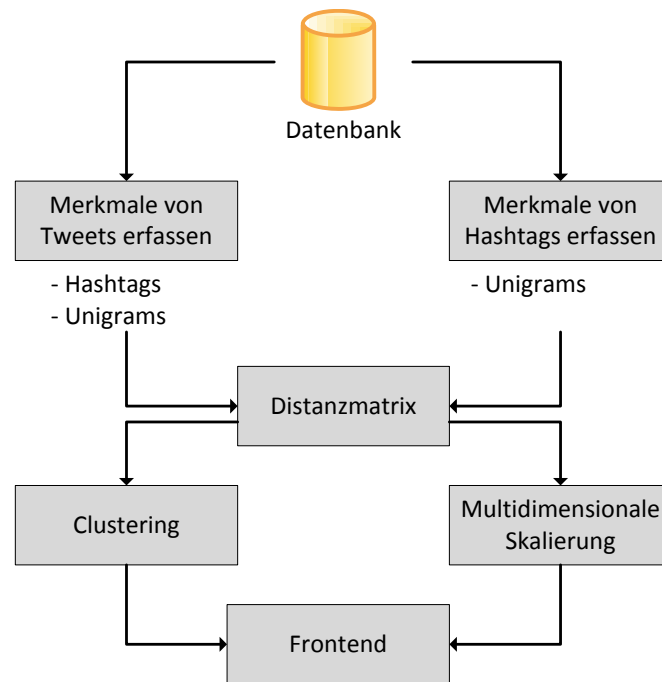


Abbildung 5.4.: Vorgehensweise bei der Cluster-Analyse

oder der Wörter eines Tweets, auch Unigrams genannt. Diese Ansätze sind in Abschnitt 5.2.1 beschrieben. Zusätzlich erfolgt eine Cluster-Analyse der Hashtags, welches Kapitel 5.2.2 beschreibt. Hier lassen sich bei der Merkmalsextraktion eines Hashtags alle Tweets, die diesen Hashtag enthalten, heranziehen. Damit sind mehr Informationen vorhanden, um einzelne Hashtags zu gruppieren. Kapitel 5.2.3 beschreibt einen zweistufigen Ansatz, welcher auf Basis der Hashtag-Cluster versucht, Tweets zu gruppieren. Anschließend wird die Distanz der Objekte basierend auf den erhobenen Merkmalen berechnet. Mit dem Distanzmaß ist es nun möglich, die eigentliche Cluster-Analyse durchzuführen (Kapitel 5.2.4). Gleichzeitig erfolgt eine Darstellung der Objekte im zweidimensionalen Raum, basierend auf den Distanzen (Kapitel 5.2.5). Die Cluster-Zuordnungen sowie die Positionen im zweidimensionalen Raum werden anschließend dem Frontend zur grafischen Aufbereitung übergeben. Das Kapitel schließt mit einer Diskussion der Ergebnisse und gibt einen Ausblick, welche Schritte in weiteren Scrum-Iterationen möglich gewesen wären.

### 5.2.1. Cluster-Analyse von Tweets

Aus Tweets lassen sich verschiedene Typen von Merkmalen ableiten, mit denen es möglich ist, Tweets nach ihrer gegenseitigen Ähnlichkeit zu bewerten. Diese Merkmalstypen werden im Folgenden anhand eines durchgehenden Beispiels erläutert:

- **Tweet 1:** #CDU #SPD Merkel ist in London
- **Tweet 2:** #CDU #SPD Merkel reist nach London
- **Tweet 3:** Merkel mag #Rom.

#### Hashtags als Merkmale

Die Erfassung der Hashtags eines Tweets stellt die erste Möglichkeit dar, Tweets gegenseitig zu bewerten. Da ein Tweet in der Regel keine Wiederholung von Hashtags enthält, zählt nur das Auftreten des Hashtags, nicht die Anzahl. Damit ist das Hashtag-Merkmal binärkodiert, wobei eine 1 für das Auftreten im Tweet steht. Im obigen Beispiel ergibt sich somit die Tabelle 5.3. Die Zeilen stellen die Tweets dar, die Spalten die erhobenen Merkmale für die Cluster-Analyse.

	#CDU	#SPD	#Rom
Tweet 1	1	1	0
Tweet 2	1	1	0
Tweet 3	0	0	1

*Tabelle 5.3.: Binäre Merkmale eines Tweets: Hashtags*

Dieses Beispiel erläutert auch die Idee hinter der Hashtag-Erfassung. Da Hashtags als Schlagwörter einen Tweet beschreiben, sollten zwei Tweets thematisch ähnlicher sein, je mehr übereinstimmende Hashtags sie besitzen. Im vorliegenden Beispiel sind sich Tweet 1 und Tweet 2 ähnlicher, denn beide verwenden die Hashtags „#CDU“ und „#SPD“.

Die Ähnlichkeitserfassung hängt von der Datenskalisierung ab. Da die Merkmale binärkodiert sind, kommen Kennzahlen in Frage, die die Übereinstimmungen abzählen. Dabei kommen jedoch nur Ähnlichkeitsmaße in Frage, bei denen das gleiche Auftreten (eine 1er-Übereinstimmung bei jeweils zwei Tweets) eine Rolle spielt. Dementsprechend wurden die Ähnlichkeitsmaße Dice, S-Koeffizient, M-Koeffizient und Russel Rao in Betracht gezogen.



Sei  $n_{11}$  die Anzahl der gemeinsam aufgetretenen Merkmale in zwei Tweets,  $n_{10}$  die Anzahl der Merkmale, die nur im ersten Tweet vorkommen,  $n_{01}$  die Anzahl der Merkmale, die nur im zweiten Tweet vorkommen und  $n_{00}$  die Anzahl der Merkmale, die in den beiden Tweets nicht vorkommen, dann sind die genannten Ähnlichkeiten zwischen zwei Tweets wie folgt definiert:

$$\frac{2n_{11}}{n_{01} + n_{10} + 2n_{11}}, \quad (\text{Dice})$$

$$\frac{n_{11}}{n_{01} + n_{10} + n_{11}}, \quad (\text{S-Koeffizient})$$

$$\frac{n_{00} + n_{11}}{n_{00} + n_{01} + n_{10} + n_{11}}, \quad (\text{M-Koeffizient})$$

$$\frac{n_{11}}{n_{00} + n_{01} + n_{10} + n_{11}}. \quad (\text{Russel Rao})$$

Die jeweiligen Kennzahlen haben einen Einfluss auf das Ergebnis der Cluster-Analyse, da sich bei jeder Kennzahl andere Ähnlichkeiten ergeben. An der Formel kann man ablesen, welche Bedeutung das jeweilige Maß hat. Verwendet man beispielsweise S-Koeffizienten, dann sind sich zwei Tweets um so ähnlicher, je mehr gemeinsame Hashtags sie haben ( $n_{11}$ ) und die Hashtags, die nur in einem der Tweets vorkommen ( $n_{01}$  und  $n_{10}$ ) reduzieren die Ähnlichkeit. Bei der Verwendung von M-Koeffizient würden im Gegensatz zu S-Koeffizient auch das Fehlen eines Hashtags in beiden Tweets ( $n_{00}$ ) sie ähnlicher machen, was natürlich nicht erwünscht ist.

Obwohl Russel Rao nur die gemeinsam auftretenden Merkmale berücksichtigt, bildet sich oft ein Cluster, der sehr viele Tweets beinhaltet. Das liegt daran, dass die Tweets zu einem gegebenen Suchbegriff von dem dazugehörigen Hashtag dominiert werden, beispielsweise beim Suchbegriff Schumacher der Hashtag „#schumacher“. Da das fast immer der Fall ist, haben wir uns gegen das Russel-Rao-Ähnlichkeitsmaß entschieden. Der M-Koeffizient berücksichtigt die Merkmale, die in beiden betrachteten Tweets nicht vorkommen ( $n_{00}$ ) und ist somit für die hier implementierte Anwendung ungeeignet.

Wir haben uns für den S-Koeffizient entschieden, weil Russel Rao und der M-Koeffizient ungeeignet sind und Dice keine signifikante Veränderungen bewirkt hat. In der praktischen Anwendung jedoch ergaben sich keine sinnvollen Distanzen, da ein Tweet nur wenig Hash-

tags enthält und somit keine oder nur wenig Übereinstimmungen mit anderen Tweets besitzt. Dadurch lassen sich keine aussagekräftigen Distanzen ableiten.

## Unigrams als Merkmale

Da die Hashtag-Erfassung keine aussagekräftigen Ergebnisse liefert, wurden als zweiten Ansatz die Unigrams (einzelne Wörter) eines Tweets als Merkmal erfasst. Da dieses Merkmal bei der Sentiment-Analyse bereits implementiert ist (siehe Kapitel 5.1), wurde hier auf diese Implementierung zurückgegriffen. Da bei der Sentiment-Analyse nur das Auftreten eines Unigrams, nicht die Anzahl der Unigrams im Tweet zählt, ergibt sich hier durch die Wiederverwendung der Implementierung erneut ein binäres Merkmal. Tabelle 5.4 zeigt das Resultat für das obige Beispiel der drei Tweets. Eine 1 steht für die Anwesenheit des Wortes im Tweet, eine 0 für die Abwesenheit. Die Reihenfolge der Merkmale in der Tabelle ergibt sich, wenn man die Tweets, beginnend mit Tweet 1, von links nach rechts durchgeht und beim ersten Auftreten eines Wortes eine neue Spalte hinzufügt. Existiert das Wort bereits als Merkmal, wird die bisherige Spalte verwendet. Außerdem ist anzumerken, dass auch Hashtags erneut als Merkmale mit einfließen, da sie ebenfalls Wörter darstellen.

	#CDU	#SPD	Merkel	ist	in	London	reist	nach	mag	#Rom
Tweet 1	1	1	1	1	1	1	0	0	0	0
Tweet 2	1	1	1	0	0	1	1	1	0	0
Tweet 3	0	0	1	0	0	0	0	0	1	1

Tabelle 5.4.: Binäre Merkmale eines Tweets: Unigrams

Das Beispiel aus der Tabelle 5.4 verdeutlicht die Idee, Unigrams als Merkmale zu verwenden. Während bei Tweet 1 und Tweet 2 vier von zehn Wörtern übereinstimmen, stimmt zum Beispiel zwischen Tweet 1 und Tweet 3 beziehungsweise Tweet 2 und Tweet 3 nur ein Wort überein. Damit sind sich Tweet 1 und Tweet 2 ähnlicher. In der konkreten Implementierung wurde der S-Koeffizient verwendet, analog zur Hashtag-Analyse.

Das Ziel durch die zusätzliche Erfassung der Unigrams eines Tweets war eine differenziertere Ähnlichkeitsberechnung, da mehr Merkmale zur Verfügung stehen. Außerdem fließen die Hashtags nach wie vor in die Analyse mit ein, da sie eine Teilmenge der erfassten Unigrams darstellen.

Allerdings ergab sich bei praktischen Anwendung das Problem, dass zwar mehr Merkmale zur Verfügung standen, aber die Tweets weiterhin nur wenig übereinstimmende

Merkmale hatten. Mit diesem Ansatz können Retweets sehr gut gruppiert und positioniert werden, da die Merkmale komplett übereinstimmen und so zu einer höheren Ähnlichkeit als zu allen anderen Tweets führen. Jedoch stimmen zwei Tweets, die keine Retweets voneinander sind, meist nur in wenigen Worten überein. Damit konnten keine aussagekräftigen Distanzen zwischen normalen Tweets berechnet werden.

Aus diesen Ergebnissen ergaben sich die folgenden beiden Erkenntnisse:

- Auch die Verwendung der Worthäufigkeiten, statt nur das bloße Auftreten eines Wortes zu verwenden, dürfte keine spürbaren Verbesserungen liefern, da Tweets aufgrund der beschränkten Länge Unigrams kaum mehrfach enthalten.
- Das Clustern von Tweets, basierend nur auf den Merkmalen aus dem Tweet-Text, ist nur schwer möglich.

In der Literatur wird meist ein zweistufiger Ansatz vorgeschlagen [39, 9]. Zunächst werden die Hashtags gruppiert und diese Ergebnisse zur Tweet-Gruppierung angewendet. Als konkreten Ansatz zur Tweet-Analyse wurde der von Tsur et al. vorgestellte Ansatz [39] implementiert. Damit ergab sich als Nebenprodukt auch eine Hashtag-Analyse, die im folgenden Kapitel vorgestellt wird. Das darauffolgende Kapitel beschreibt die Cluster-Analyse basierend auf den Hashtag-Gruppen.

### 5.2.2. Cluster-Analyse von Hashtags

Während ein einzelner Tweet nur wenige Merkmale auf Grund der beschränkten Länge bietet, lässt sich dieser Flaschenhals bei der Hashtag-Gruppierung beseitigen. Wie in [39] beschrieben, werden zu einem bestimmten Hashtag alle Tweets, die diesen enthalten, zur Merkmalserfassung verwendet. Damit ist die Textlänge wesentlich größer als die bei der Merkmalserfassung eines einzelnen Tweets. Die Idee ist also, dass nicht nur mehr Merkmale zur Verfügung stehen, sondern auch, dass durch die größere Textlänge mehr Merkmale einen von Null verschiedenen Wert besitzen. Dadurch sollten sich aussagekräftigere Distanzen ableiten können.

Die einzelnen Schritte lassen sich wie folgt beschreiben:

1. Rufe zu einem Suchbegriff eine bestimmte Menge an Tweets ab. Diese Menge stellt einen Trade-Off zwischen der Genauigkeit und der Laufzeit dar. Eine hohe Anzahl

verbessert die Genauigkeit der Ergebnisse, da mit mehr Tweets auch mehr Informationen zur Verfügung stehen. Allerdings führt dies zu einer längeren Wartezeit im Frontend bis zur Anzeige der Ergebnisse.

2. Erstelle für jeden Hashtag, der in der Tweet-Menge auftritt, ein virtuelles Dokument. Ein virtuelles Dokument zu einem Hashtag ist eine Konkatenation aller Tweet-Texte, die diesen Hashtag enthalten [39]. Ein Tweet kann durchaus für mehrere Hashtags verwendet werden. Abbildung 5.5 zeigt diesen Schritt für das obige Beispiel der drei Tweets.

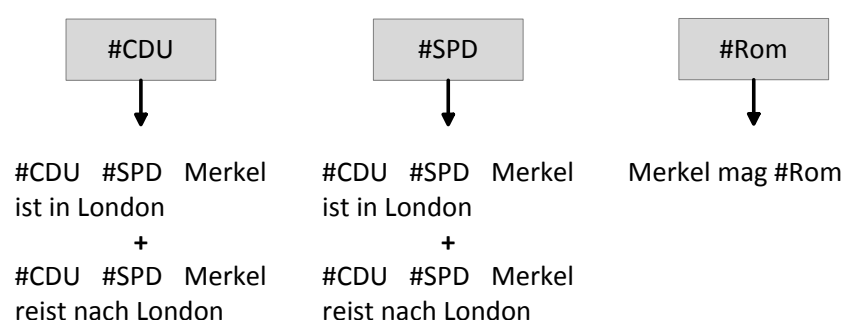


Abbildung 5.5.: Erzeugung eines jeweiligen virtuellen Dokuments zu einem Hashtag

3. Entferne alle Hashtags, die weniger als  $x$  Tweets enthalten. Verschiedene Werte für  $x$  wurden ausprobiert und mit  $x = 15$  gute Ergebnisse erzielt. Dieser Korrekturschritt ist notwendig, damit seltene Hashtags aus der Analyse fallen. Enthält ein Hashtag beispielsweise nur zwei oder drei Tweets, tritt nach wie vor die Problematik der vorherigen Tweet-Analyse auf. Es lassen sich nur wenig aussagekräftige Merkmale ableiten. In Abbildung 5.5 würde zum Beispiel „#Rom“ mit  $x = 2$  entfernt werden.
4. Berechne die Merkmale für jeden Hashtag. Anders als bei der Tweet-Analyse, werden nun die Worthäufigkeiten erfasst. Während ein Wort bei einem einzelnen Tweet in der Regel nur selten wiederholt wird, kann die Wiederholung bei einem virtuellen Dokument in anderen Größenordnungen auftreten. Daher lohnt es sich hier, als Merkmal die Worthäufigkeit und nicht mehr das bloße Auftreten eines Wortes zu erfassen. Tabelle 5.5 zeigt die Merkmale für die jeweiligen Hashtags aus dem obigen Beispiel, wobei keine Hashtags entfernt werden.

	#CDU	#SPD	Merkel	ist	in	London	reist	nach	mag	#Rom
#CDU	2	2	2	1	1	2	1	1	0	0
#SPD	2	2	2	1	1	2	1	1	0	0
#Rom	0	0	1	0	0	0	0	0	1	1

Tabelle 5.5.: Quantitative Merkmale eines Hashtags: Worthäufigkeiten

5. Dividiere jedes Merkmal eines Hashtags durch die entsprechende Zeilensumme. Die Zeilensumme stellt die Gesamtanzahl der Wörter eines Hashtags dar. Durch die Division der Häufigkeit eines Wortes durch die Gesamtanzahl ergibt sich ein relativer Anteil dieses Wortes an der Gesamtzahl der verwendeten Wörter. Im Beispiel ergibt sich somit für das Merkmal „Merkel“ des Hashtags „#CDU“ der Wert  $2/12$ .

Diese Datenvorbereitung ist notwendig, da jedes virtuelle Dokument aus einer anderen Anzahl an Tweets gebildet wird. Ist zum Beispiel das Wort „kerry“ in 20 von 40 Wörtern beim Hashtag „#merkel“ vorhanden und in 10 von 20 Wörtern beim Hashtag „#steinmeier“, so unterscheiden sich beide Hashtags bei der absoluten Wortanzahl deutlich. Relativ gesehen tritt „kerry“ aber bei beiden Hashtags in 50% der Wörter auf. Beide Hashtags sind sich also ähnlicher.

6. Berechne die Distanzmatrix. Da es sich bei der Wortanzahl um ein quantitatives Merkmal handelt, lassen sich nun andere Distanzmaße verwenden. Hier wurde die euklidische Distanz als Metrik eingesetzt. Weitere Scrum-Iterationen sollten auch die Evaluierung anderer Metriken beinhalten.

7. Führe einen Cluster-Algorithmus sowie eine multidimensionale Skalierung durch.

Das beschriebene Vorgehen basiert auf [39], ist aber mit Schritt 3 und 5 sowie mit der multidimensionalen Skalierung aus Schritt 7 entsprechend erweitert.

In der praktischen Anwendung ergaben sich deutlich bessere Ergebnisse als bei der vorherigen Tweet-Analyse, da die Distanzen auf Grund der mehr zur Verfügung stehenden Informationen aussagekräftiger sind. So wurden zum Beispiel „#merkel“, „#schumacher“ und „#ski“ (Stand: 12.03.2014, „Alle Sprachen“) in einem Cluster dicht aneinander dargestellt. Daraus ließ sich zum Beispiel mit dem News-Modul ableiten, dass viele Tweets den Skiunfall von Merkel und Schumacher behandeln.

Schwierig ist jedoch die Interpretation im Frontend, warum zwei Hashtags in einem Cluster liegen. Künftige Scrum-Iterationen sollten daher im Frontend zum Beispiel eine

Möglichkeit bieten, die Tweets zu zwei markierten Hashtags anzuzeigen. Damit lässt sich besser ein Gefühl vermitteln, wie das Ergebnis zustande gekommen ist.

### 5.2.3. Verwendung der Hashtag-Cluster zur Tweet-Gruppierung

Sind die Hashtag-Cluster aus einer Menge an Tweets berechnet, lassen sich nun im weiteren Schritt die Tweet-Gruppen berechnen. Dazu werden die Tweets, die das jeweilige virtuelle Dokument zu einem Hashtag bilden, in den gleichen Cluster eingeordnet wie der entsprechende Hashtag [39]. Angenommen, im obigen Beispiel erhält der Hashtag „#Rom“ die Cluster-Nummer 2. Alle Tweets, die im virtuellen Dokument von „#Rom“ liegen, erhalten somit ebenfalls die Cluster-Nummer 2.

Ein Problem ist allerdings, dass ein Tweet durchaus in mehreren virtuellen Dokumenten enthalten sein kann, sofern mehrere Hashtags verwendet werden. Besitzen diese unterschiedliche Cluster-Zugehörigkeiten, ist eine 1:1-Zuordnung zwischen Hashtag-Cluster und Tweet-Cluster nicht mehr möglich.

Daher wurde versucht, für jeden Tweet gemäß einer Distanzfunktion den nächsten Hashtag zu finden. Der Tweet erbt anschließend die Cluster-Nummer des Hashtags. Damit ist zwar eine eindeutige 1:1-Zuordnung möglich, aber es entsteht erneut das Problem, dass ein Tweet selbst zu wenig Merkmale enthält, um aussagekräftige Distanzen zu den Hashtags abzuleiten.

Ein weiteres Problem des Ansatzes ist die Anforderung, Tweets durch die multidimensionale Skalierung in die 2D-Ebene einzubetten. Ein Tweet selbst erhält nur die Cluster-Zugehörigkeit des Hashtags, hat dadurch aber selbst keine Distanzen mehr zu anderen Tweets. Alle Tweets würden daher in einem Punkt dargestellt werden, dem Punkt, an dem theoretisch der Hashtag liegen würde. Damit lassen sich aber die Distanzen zwischen den Tweets nicht mehr interpretieren. Da dies nicht die ursprüngliche Motivation widerspiegelt, wurde das Ergebnis des hier geschilderten Ansatzes nicht im Frontend dargestellt, ist aber nach wie vor implementiert.

### 5.2.4. Cluster-Algorithmen

Bei der Wahl der Cluster-Algorithmen haben wir uns auf hierarchische Clusterverfahren beschränkt, die ohne Koordinaten der einzelnen Punkte auskommen. Der Grund dafür ist, dass man den einzelnen Entitäten, in unserem Fall Tweets und Hashtags, zunächst keine Positionen zuordnen kann, d. h. die Tweets beziehungsweise Hashtags haben zwar Ähn-

lichkeiten zu einander, die man als Distanzen interpretieren kann, aber keine festgelegten Positionen in Raum. Es wäre zwar möglich gewesen, zuerst die multidimensionale Skalierung durchzuführen und anschließend mit Hilfe der gewonnen Positionen zu clustern, aber es ist sehr unwahrscheinlich, dass eine größere Anzahl an Entitäten mit geringen Fehlern in die 2D-Ebene eingebettet werden können. Partitionierende Clusteringverfahren, wie  $k$ -Means [23] kamen aus diesem Grund nicht in Frage.

Es sind zwei hierarchische Cluster-Algorithmen implementiert worden, ein Single-Linkage-Algorithmus und ein Complete-Linkage-Algorithmus. Durch die Implementierung beider Algorithmen ist es möglich, die Auswirkungen beider Ansätze auf das Ergebnis zu beobachten. Da Complete Linkage kompaktere Cluster erzeugt, haben wir uns für diesen Algorithmus entschieden. Im Folgenden werden die beiden Cluster-Algorithmen einzeln vorgestellt.

### Single-Linkage-Algorithmus

Die Cluster-Analyse lässt sich auch als Graph-Problem darstellen, wobei die Knoten die zu gruppierenden Objekte darstellen (hier Tweets oder Hashtags). Außerdem handelt es sich um einen vollständigen Graphen, da alle möglichen Objektpaare aus der Distanzmatrix abgebildet werden. Die Kantenkosten sind die jeweiligen Distanzen aus der Distanzmatrix.

Das Single-Linkage-Clustering stellt nun das gleiche Problem dar wie die Berechnung des minimalen Spannbaums (MST) durch den Kruskal-Algorithmus [20]. Jede hinzugefügte Kante bei der MST-Berechnung stellt einen Fusionierungsschritt beim Clustern dar. Sollen beispielsweise  $N$  Cluster berechnet werden, muss bei der Verwendung einer Union-Find-Datenstruktur abgebrochen werden, sobald  $N$  Repräsentanten übrig sind.

Der Kruskal-Algorithmus besitzt eine Laufzeit von  $\mathcal{O}(E \log E)$ , wobei  $E$  die Kantenanzahl darstellt. Da hier ein vollständiger Graph abgearbeitet wird, gilt hier  $E = M^2$  mit  $M$  als die Anzahl der zu clusternden Entitäten (Knotenanzahl). Durch die Adaption des Kruskal-Algorithmus besitzt der Single-Linkage-Algorithmus damit eine Laufzeit von  $\mathcal{O}(M^2 \log M^2) = \mathcal{O}(M^2 \log M)$ .

### Complete-Linkage-Algorithmus

Die Idee bei der Implementierung des Complete-Linkage-Algorithmus war, den naiven hierarchischen Algorithmus zu verwenden, an die eigenen Bedürfnisse anzupassen und anschließend die Laufzeit zu verbessern.

Sei  $S = \{t_1, \dots, t_M\}$  die Menge der zu clusternden Entitäten,  $d : S^2 \rightarrow \mathbb{R}$  eine Distanzfunktion und  $N$  die maximale Anzahl der Cluster, dann lässt sich der implementierte hierarchische Algorithmus zu Algorithmus 2 zusammenfassen.

---

**Algorithmus 2** Generischer hierarchischer Algorithmus

---

**Eingabe:** Menge der Entitäten  $\{t_1, \dots, t_M\}$ ,  $N$  die maximale Anzahl der Cluster  
**Ausgabe:** Menge der Cluster  $\mathcal{C}$

```

1:  $\mathcal{C} = \{\{t_1\}, \{t_2\}, \dots, \{t_M\}\}$ 
2: for ( $i = 1; i < M$  and  $|\mathcal{C}| > N; i++$ ) do
3:    $(c_i, c_j) = \arg \min_{c_i, c_j \in \mathcal{C}, i \neq j} d(c_i, c_j)$ 
4:    $\mathcal{C} = (\mathcal{C} \setminus \{c_i, c_j\}) \cup \{c_i \cup c_j\}$ 
5: end for
```

---

Die Wahl der Funktion  $d$  bestimmt das Verhalten des Algorithmus. Wir haben  $d$  entsprechend dem Complete-Linkage wie folgt gewählt:

$$d(c_i, c_j) := \max \{\delta(t_i, t_j) \mid t_i \in c_i, t_j \in c_j\}.$$

Dabei stellt der Ausdruck  $\delta(t_i, t_j)$  die aus der Distanzmatrix stammenden Unähnlichkeiten zwischen den Tweets beziehungsweise Hashtags dar.

Bei der naiven Implementierung des obigen Algorithmus beträgt die Laufzeit  $\mathcal{O}(M^3)$ . Um die Laufzeit auf  $\mathcal{O}(M^2 \log M)$  zu verbessern, haben wir den Algorithmus so modifiziert, dass zur Verwaltung der Unähnlichkeiten ein Heap verwendet wird. Insgesamt müssen durch den Heap  $\frac{M^2}{2} - M \in \mathcal{O}(M^2)$  Werte verwaltet werden. Es werden aber auch maximal  $\frac{M^2}{2} - M$  Werte wieder aus dem Heap entfernt, falls  $N = 1$  gilt (*worst case*). Insgesamt wird also zum Erzeugen und Verwalten des Heaps  $\mathcal{O}(M^2 + M^2 \log M^2) = \mathcal{O}(M^2 \log M)$  Zeit benötigt, was auch der Laufzeit von dem implementierten Algorithmus entspricht.

### 5.2.5. Multidimensionale Skalierung

Das Ziel der multidimensionalen Skalierung ist es, die gegebenen Objekte im Raum oder in der Ebene anzuordnen. Dabei sollen die Abstände der Objekte zueinander nach der multidimensionalen Skalierung ihre Unähnlichkeiten möglichst gut wiedergeben. Dadurch ist es möglich, Objekte, die eigentlich keine Punkte im euklidischen Raum sind, als solche zu interpretieren und so Informationen anschaulicher darzustellen oder neue Zusammenhänge



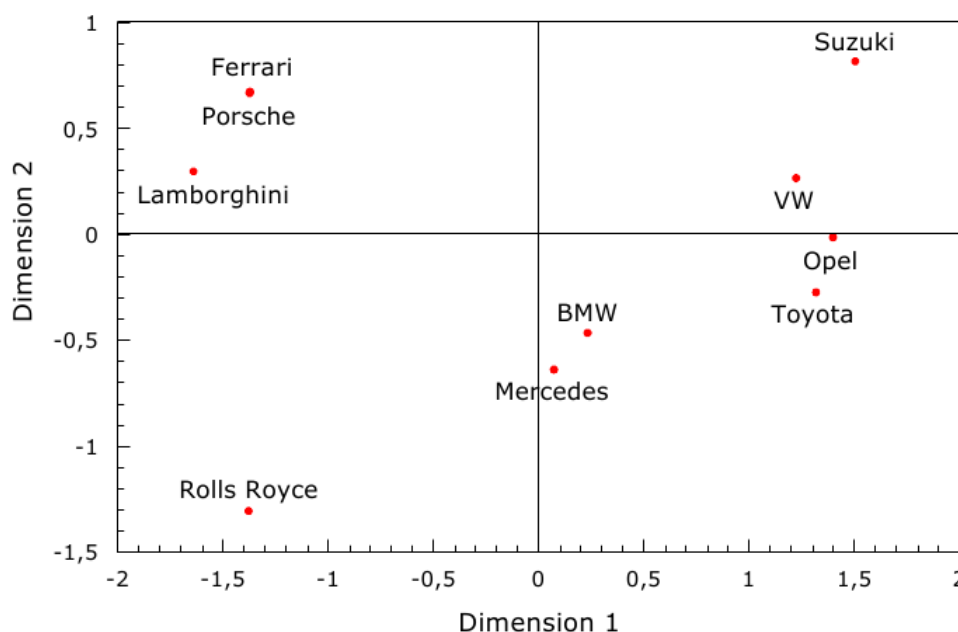


Abbildung 5.6.: Multidimensionale Skalierung angewandt auf eine Umfrage zu Unterschieden zwischen bekannten Automarken. Bild entnommen aus [19].

zu entdecken. In der Abbildung 5.6 ist die Anwendung der multidimensionalen Skalierung auf eine Umfrage zu unterschiedlichen Automarken dargestellt. Bei der Umfrage haben die Befragten angegeben, wie ähnlich die betroffenen Automarken zueinander sind. Man kann sehen, dass bei der geeigneten Wahl von Parametern und Verfahren die Punkte in bestimmten Teilen der Ebene liegen, die charakteristisch für sie sind, zum Beispiel liegen die Sportwagenhersteller nah bei einander.

Für unser Projekt haben wir zwei Verfahren zur multidimensionalen Skalierung implementiert: *classical scaling* und den Smacof-Algorithmus. Beide Verfahren sind in [12] ausführlich beschrieben. *Classical scaling* ist ein Verfahren, das analytisch die multidimensionale Skalierung berechnet, wohingegen der Smacof-Algorithmus ein iteratives Verfahren ist und wesentlich mehr Parameter bietet. In den folgenden Kapiteln werden die beiden Verfahren kurz vorgestellt. Für eine ausführliche Beschreibung der vorgestellten Verfahren verweisen wir auf [12].

## Classical Scaling

*Classical scaling* berechnet analytisch zu gegebenen Unähnlichkeiten, die als Distanzen interpretiert werden, eine passende Punktemenge. Die Abstände der Punkte untereinander

entsprechen den gegebenen Distanzen, wenn die Dimension des Raumes, in dem die Punkte liegen, hoch genug ist. Der Sinn und Zweck der multidimensionalen Skalierung ist es, die Dimension, in der die Punkte liegen, gering zu halten. Damit lassen sich diese grafisch darstellen, wenn die Dimension kleiner als vier ist. Aus diesem Grund können die berechneten Punkte die vorgegebenen Distanzen nur approximieren. Wobei die Approximation schlechter wird, wenn die Anzahl der Dimensionen sinkt.

Sei  $\Delta \in \mathbb{R}^{n \times n}$  die Matrix mit den Distanzen der  $n$  Punkte. Ziel ist es eine (von vielen) Punktekongstellations zu finden, die diese Matrix erzeugt und eine Approximation davon in einem niedrigdimensionalen Raum zu nehmen. Das Vorgehen bei *classical scaling* lässt sich dann wie folgt zusammenfassen:

1. Berechne  $\Delta^{(2)} = \text{Quadrate der Einträge von } \Delta$ .
2. Berechne  $-\frac{1}{2}\mathbf{C}_n\Delta^{(2)}\mathbf{C}_n =: \mathbf{X}\mathbf{X}^T$ ,  $\mathbf{C}_n = \mathbf{I}_n - \frac{1}{n}\mathbf{1}_n$ ,  $\mathbf{I}_n$  ist die  $n$ -dimensionale Einheitsmatrix und  $\mathbf{1}_n$  ist die  $n$ -dimensionale Matrix mit Einsen.
3. Berechne die Eigenwertdekomposition  $\mathbf{X}\mathbf{X}^T = \mathbf{Q}\mathbf{\Lambda}\mathbf{Q}^T = \mathbf{Q}\mathbf{\Lambda}^{1/2}(\mathbf{Q}\mathbf{\Lambda}^{1/2})^T$ ,  $\mathbf{\Lambda}^{1/2}$  ist Diagonalmatrix mit Quadratwurzeln der Elemente aus  $\mathbf{\Lambda}$ .
4. Berechne für ein gewünschtes  $d \in \{1, 2, 3\}$ :  $\mathbf{X}_d = \mathbf{Q}_d\mathbf{\Lambda}_d^{1/2}$ , dabei ist  $\mathbf{\Lambda}_d^{1/2}$  die Matrix mit den Quadratwurzeln der  $d$  größten positiven Eigenwerte und  $\mathbf{Q}_d$  ist die Matrix mit entsprechenden Eigenvektoren.

Erklärung der Schritte:

1. Nimmt man an, dass man Punkte hat, die die Distanzen aus der Matrix  $\Delta$  erzeugen und die Koordinaten dieser Punkte stehen in der Matrix  $\mathbf{X}$  (Zeilenweise), dann kann man den quadrierten euklidische Abstand zweier Vektoren, wie folgt berechnen:

$$d_{ij}^2 = \sum_{k=1}^n (x_{ik} - x_{jk})^2 = \sum_{k=1}^n (x_{ik}^2 - 2x_{ik}x_{jk} + x_{jk}^2). \quad (5.1)$$

Berechnet man die quadrierten euklidischen Abstände für alle Paare, erhält man die Einträge der Matrix  $\Delta^{(2)}$ . Man rechnet mit den quadrierten Abständen, weil man dadurch die Summe aus (5.1) in mehrere Summen aufteilen kann. Dadurch kann man die Matrix  $\Delta^{(2)}$  wie folgt schreiben:

$$\Delta^{(2)}(\mathbf{X}) = \mathbf{c}\mathbf{1}^T + \mathbf{1}\mathbf{c}^T - 2\mathbf{X}\mathbf{X}^T,$$

wobei  $\mathbf{c}$  ein Vektor mit den Diagonalelementen aus  $\mathbf{XX}^T$  und  $\mathbf{1} \in \mathbb{R}^n$  ist ein Vektor mit Einsen sind. Dieser Zusammenhang ist der Grund für die Berechnung von  $\Delta^{(2)}$ .

2. In diesem Schritt wird die Matrix  $\Delta^{(2)}$  beidseitig mit der Matrix  $\mathbf{C}_n$  multipliziert ( $\mathbf{C}_n$ , wie oben in Schritt 2). Matrix  $\mathbf{C}_n$  ist die Zentrierungsmatrix, d. h. eine Anwendung auf einen Vektor bewirkt das Subtrahieren des Mittelwertes seiner Einträge von jedem Eintrag, z. B.

$$\mathbf{C}_2 \begin{pmatrix} 2 \\ 4 \end{pmatrix} = \begin{pmatrix} \frac{1}{2} & -\frac{1}{2} \\ -\frac{1}{2} & \frac{1}{2} \end{pmatrix} \begin{pmatrix} 2 \\ 4 \end{pmatrix} = \begin{pmatrix} -1 \\ 1 \end{pmatrix}.$$

Wichtige Spezialfälle sind:

- Zentrierung eines Vektors, der nur Einsen enthält und
- Zentrierung eines bereits zentrierten Vektors.

Im ersten Fall erhält man den Nullvektor, weil der Mittelwert eines Vektors nur mit Einsen immer den Wert 1 hat. Im zweiten Fall bewirkt die Zentrierung nichts, da der Vektor bereits zentriert ist. Multipliziert man  $\Delta^{(2)}$  beidseitig mit  $\mathbf{C}_n$ , ergibt sich:

$$\begin{aligned} \mathbf{C}_n \Delta^{(2)} \mathbf{C}_n &= \mathbf{C}_n (\mathbf{c} \mathbf{1}^T + \mathbf{1} \mathbf{c}^T - 2 \mathbf{X} \mathbf{X}^T) \mathbf{C}_n \\ &= \mathbf{C}_n \mathbf{c} \mathbf{1}^T \mathbf{C}_n + \mathbf{C}_n \mathbf{1} \mathbf{c}^T \mathbf{C}_n - 2 \mathbf{C}_n \mathbf{X} \mathbf{X}^T \mathbf{C}_n \\ &= \mathbf{C}_n \mathbf{c} \mathbf{0}^T + \mathbf{0} \mathbf{c}^T \mathbf{C}_n - 2 \mathbf{C}_n \mathbf{X} \mathbf{X}^T \mathbf{C}_n \\ &= \underbrace{-2 \mathbf{C}_n \mathbf{X} \mathbf{X}^T \mathbf{C}_n}_{\text{X ist zentriert}} \\ &= -2 \mathbf{X} \mathbf{X}^T. \end{aligned}$$

Im letzten Schritt der Rechnung nimmt man an, dass die Matrix  $\mathbf{X}$  zentriert ist. Das ist keine Einschränkung, weil man eine beliebige Lösung sucht. Außerdem lässt sich die Lösung durch eine Translation anpassen, sodass die Distanzen zwischen den Punkten unverändert bleiben.

3. Um aus der Matrix  $\mathbf{XX}^T$  die Matrix  $\mathbf{X}$  zu extrahieren, benutzt man die Eigenwertdekomposition. Dabei macht man die Annahme, dass die Eigenwerte nicht negativ sind.

4. Um eine Approximation von  $\mathbf{X}$  für die gewünschte Dimension zu erhalten, benutzt man die größten Eigenwerte und die entsprechenden Eigenvektoren.

Es ist wichtig anzumerken, dass *classical scaling* mit der Hauptkomponentenanalyse [18] eng verwandt ist. Der Unterschied ist hilfreich für das Verständnis des Vorgehens beim *classical scaling*. Bei der Hauptkomponentenanalyse möchte man die Dimension der betrachteten Punkte reduzieren, während es bei *classical scaling* zusätzlich darum geht, die Punkte selbst zu finden.

### Smacof-Algorithmus

Die Idee beim Smacof-Algorithmus ist, eine gewählte Funktion  $\sigma_{\Delta} : \mathbb{R}^{n \times d} \rightarrow \mathbb{R}$  zu minimieren. Die Funktion  $\sigma_{\Delta}$  ist dabei ein Maß für den Fehler, der bei der Wahl der  $n$  Punkte im  $d$ -dimensionalen Raum entsteht. Wir haben in unserer Implementierung  $\sigma_{\Delta}$  wie folgt gewählt:

$$\sigma_{\Delta}(\mathbf{X}) = \sum_{i=1}^n \sum_{j=i+1}^n (d(\mathbf{x}_i, \mathbf{x}_j) - \delta_{ij})^2.$$

Dabei ist  $\mathbf{x}_i$  die  $i$ -te Zeile von  $\mathbf{X}$ ,  $d(\mathbf{x}_i, \mathbf{x}_j) = \|\mathbf{x}_i^T - \mathbf{x}_j^T\|$ ,  $\Delta$  ist die Unähnlichkeitsmatrix (die Einträge an der Position  $ij$  geben an, wie unähnlich Entität  $i$  zur Entität  $j$  ist) und  $\delta_{ij}$ 's sind die Einträge von  $\Delta$ . Die Lösung soll also eine Punktmenge sein, sodass die Summe der Quadrate der Differenzen zwischen den resultierenden Abständen und Unähnlichkeiten minimiert wird.

Zur Initialisierung haben wir die Punkte entsprechend ihrer Clusterzugehörigkeit im Kreis angeordnet, sodass sie um ihre Clustercentren zufällig verteilt sind, siehe Abbildung 5.7. Als Alternative kann man die Ausgabe von *classical scaling* verwenden oder man startet mit einer zufälligen Konfiguration. Da in unserem Fall aber eine Clusterzugehörigkeit eine wichtige Rolle spielt, haben wir uns für die erste Variante entschieden.

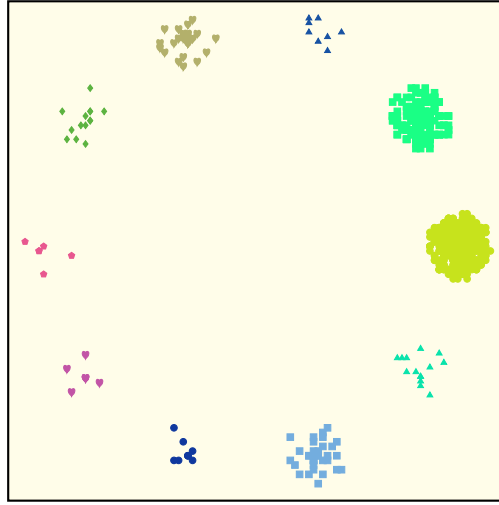


Abbildung 5.7.: Initialisierung des Smacof Algorithmus.

Der implementierte Smacof-Algorithmus ist im Algorithmus 3 zusammengefasst. Der wichtigste Schritt in dem Algorithmus ist der Minimierungsschritt in Zeile 5. Die Matrix  $\mathbf{B}_\Delta(\mathbf{X})$  hat dabei folgende Einträge:

$$b_{ij} = \begin{cases} -\frac{\delta_{ij}}{\|\mathbf{x}_i^T - \mathbf{x}_j^T\|}, & \text{falls } i \neq j \text{ und } \|\mathbf{x}_i^T - \mathbf{x}_j^T\| \neq 0, \\ 0, & \text{falls } i \neq j \text{ und } \|\mathbf{x}_i^T - \mathbf{x}_j^T\| = 0, \end{cases}$$

$$b_{ii} = -\sum_{j=1, j \neq i}^n b_{ij}.$$

---

### Algorithmus 3 Smacof Algorithmus

---

**Eingabe:**  $\Delta \in \mathbf{R}^{n \times n}$  Unähnlichkeitsmatrix mit den Einträgen  $\delta_{ij}$

**Ausgabe:**  $\mathbf{X} \in \mathbf{R}^{n \times d}$  Koordinatenmatrix.

- 1:  $\mathbf{X}$  = Initialpositionen
  - 2:  $\sigma^{\text{pre}} = \infty$
  - 3:  $\sigma^{\text{act}} = \sigma_\Delta(\mathbf{X})$
  - 4: **for** ( $i = 0; i < N_{\text{max}}$  **and**  $(\sigma^{\text{pre}} - \sigma^{\text{act}}) > \epsilon; i++$ ) **do**
  - 5:      $\mathbf{X} = \frac{1}{n} \mathbf{B}_\Delta(\mathbf{X}) \mathbf{X}$
  - 6:      $\sigma^{\text{pre}} = \sigma^{\text{act}}$
  - 7:      $\sigma^{\text{act}} = \sigma_\Delta(\mathbf{X})$
  - 8: **end for**
-

In jeder Schleifeniteration wird der Fehler  $\sigma^{\text{act}}$  kleiner. Der Algorithmus stoppt, wenn die Fehlerdifferenz  $(\sigma^{\text{pre}} - \sigma^{\text{act}})$  kleiner ist als eine festgelegt Konstante  $\epsilon$  oder die maximale Anzahl der Iterationen  $N_{\text{max}}$  erreicht wurde.

### 5.2.6. Diskussion

Die Identifizierung von Tweet-Gruppen, die die selbe Meinung zu einem Thema vertreten, stellte die Motivation der Cluster-Analyse dar. Die zusätzliche Darstellung der Tweets in einer 2D-Ebene sollte die Darstellung dieser Gruppen erleichtern. Distanzen zwischen den Tweets sollten widerspiegeln, wie ähnlich die Meinungen und Themen sind.

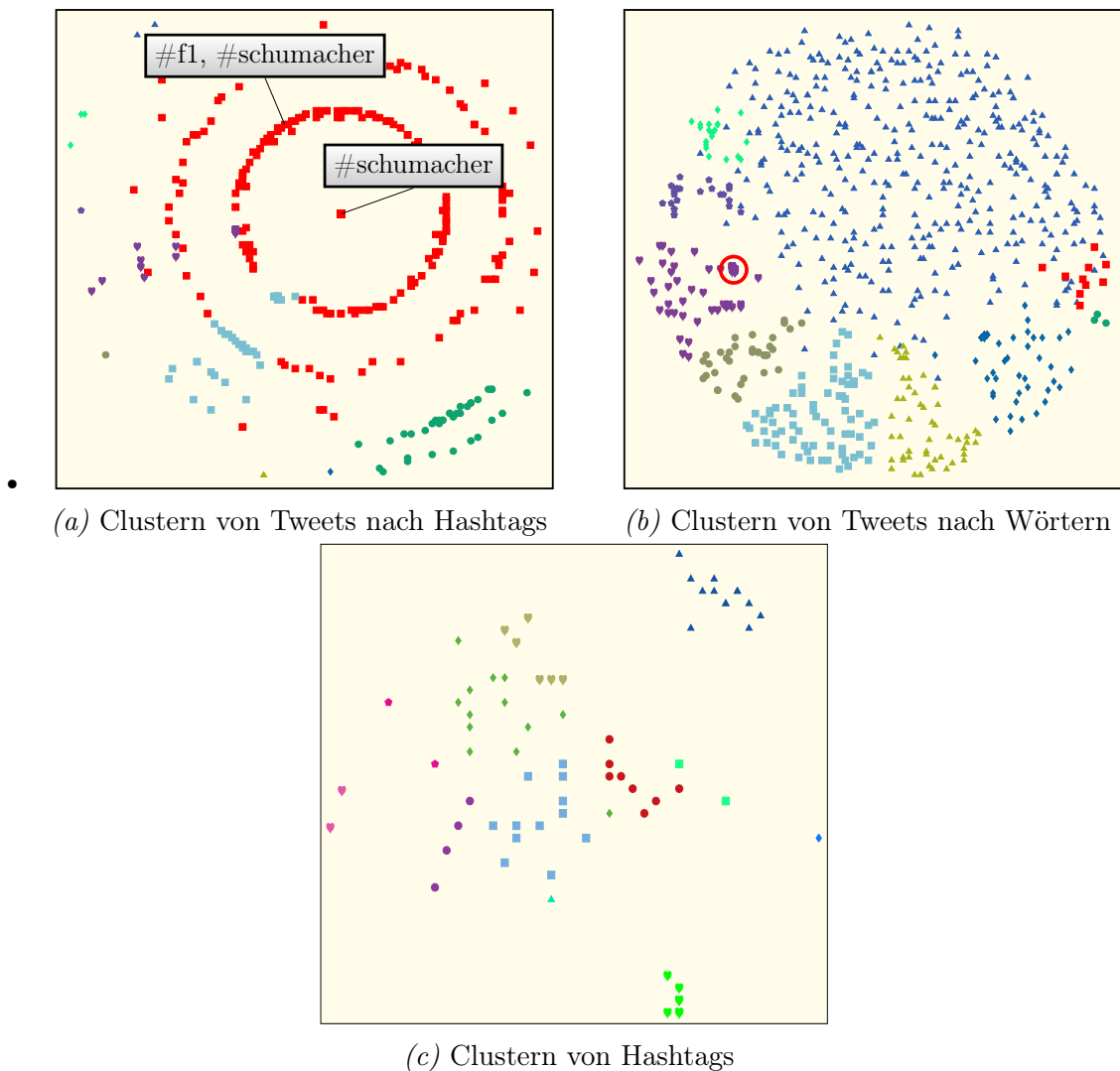


Abbildung 5.8.: Ergebnisse der Clusteranalyse

Während der Implementierung zeigte sich, dass die beschränkte Länge eines Tweets das größte Problem darstellt. Es lassen sich zu wenig aussagekräftige Merkmale aus einem einzelnen Tweet extrahieren. Bei der Verwendung von Hashtags zur Cluster-Analyse von Tweets sind die Distanzen zwischen den Tweets nicht gleichmäßig verteilt. Wenn sich zwei Tweets um einen Hashtag unterscheiden, entstehen entsprechende Sprünge in den Ähnlichkeiten. Die Anzahl der Hashtags ist viel kleiner als die Anzahl der Tweets (oft max. 100 Hashtags vorhanden), dadurch sind die Sprünge deutlich zu sehen. Abbildung 5.8a zeigt das entsprechende Resultat.

Mit der Verwendung der Unigrams der Tweets werden zwar alle Wörter zur Analyse herangezogen und es stehen mehr Merkmale zur Ähnlichkeitsanalyse zur Verfügung, jedoch stimmen zwischen zwei Tweets meist nur wenige Merkmale überein. Wie Abbildung 5.8b zeigt, sind die Tweets feiner verteilt. Allerdings lassen sich nach wie vor keine sinnvollen Ähnlichkeiten zwischen zwei Tweets berechnen. Die ideale Situation, in der dieser Ansatz funktioniert, stellen Retweets dar. Diese liegen in einem Cluster dicht aneinander, da alle Unigrams übereinstimmen (mit einem roten Kreis in der Abbildung 5.8b markiert). Da dieses Verfahren feinere Distanzen zwischen den Tweet-Gruppen liefert, stellt das Frontend die Cluster-Ergebnisse durch die Unigram-Merkmale dar.

Diese Ergebnisse motivierten einen zweistufigen Ansatz, welchen Tsur et al. in [39] vorstellen. Zunächst erfolgt die Identifizierung von Hashtag-Gruppen. Bei der Merkmals-erfassung eines Hashtags werden alle Tweets verwendet, die diesen Hashtag enthalten. Aufgrund der größeren Textlänge lohnt es sich nun, die Häufigkeit eines jeden Wortes als Merkmal zu erfassen. Damit sind die Distanzen aussagekräftiger. Anschließend erhält ein jeweiliger Tweet die Hashtag-Gruppe, zu der er am ähnlichsten ist. Damit besitzen aber die Tweets selbst keine Distanzen zueinander, womit auch die ursprüngliche Motivation verletzt ist, dass die Distanzen in der 2D-Projektion die thematische Ähnlichkeit widerspiegeln. Daher stellt das Frontend diesen Ansatz zunächst nicht dar.

Da allerdings die Hashtag-Cluster sehr viel aussagekräftiger sind, wurde dieser Ansatz mit in das Frontend als eigenständiges Verfahren zusätzlich zur Tweet-Gruppierung übernommen. Hiermit lassen sich thematisch zusammengehörige Hashtags identifizieren. Je näher zwei Hashtags in der 2D-Projektion liegen, desto ähnlicher sind sich auch die Tweets, die diese Hashtags enthalten. Abbildung 5.8c zeigt ein Ergebnis der Hashtag-Gruppierung.

Während also die Hashtag-Gruppierung funktioniert, sind für die Tweet-Gruppierung neue Ansätze notwendig. Zum einen wären String-Matching-Algorithmen, genauer Ver-

fahren zum „*approximate string matching*“, möglich, sodass mehr Merkmale zwischen den Tweets übereinstimmen. Damit könnten Unigrams auch trotz einer leicht unterschiedlichen Zeichenfolge übereinstimmen, wie zum Beispiel bei „#merkel“ und „#merkels“. Zum anderen kann versucht werden, beim zweistufigen Ansatz aussagekräftigere Distanzen auch für die Tweets abzuleiten. Beispielsweise könnte separat eine Distanzberechnung für alle Tweets in einem Hashtag-Cluster erfolgen. Hier könnten mehr Merkmale zwischen zwei einzelnen Tweets übereinstimmen, da die Tweets thematisch vorselektiert sind. Damit geben die Hashtag-Cluster global die Distanzen zwischen den Themen an. Die lokalen Distanzen innerhalb eines Clusters geben die Ähnlichkeit zwischen den Tweets wider.



## 5.3. Nachrichtenmodul

Im Laufe des Projektseminars kam der Kundenwunsch auf, bei der Betrachtung des Graphen, welcher die Tweets pro Stunde (TPS) darstellt, zusätzliche Informationen zu erhalten. Diese sollen erklären, warum die Aktivität zu manchen Zeitpunkten außerordentlich hoch ist. Daraufhin wurde entschieden, ein Modul zu entwickeln, das dem Nutzer Nachrichten zu relevanten Zeitpunkten automatisch anzeigen kann.

Hinsichtlich der Benutzerfreundlichkeit und Nutzerbindung ist diese Funktion sinnvoll, da ein Nutzer die dargestellten Daten besser interpretieren kann, ohne dabei auf andere Systeme zurückgreifen zu müssen.

### 5.3.1. Identifikation interessanter Zeitpunkte

Um Zeitpunkte auf dem TPS-Graphen zu finden, die für einen Nutzer von Interesse sein könnten, wurden typische Methoden der Signalverarbeitung zur Suche nach Peaks evaluiert.

Eine der einfachsten Methoden ist es, jeden Zeitpunkt als Peak zu markieren, der das absolute Maximum seiner unmittelbaren Nachbarschaft ist. Wie in Abbildung 5.9 zu sehen ist, erzielt dies aufgrund der häufigen Schwankungen kein akzeptables Ergebnis. Statt der unmittelbaren Nachbarschaft ein Fenster fester Breite um den zu untersuchenden Punkt zu betrachten, erzeugt ebenfalls keine Peakauswahl, die der Intuition eines Benutzers entspricht. Wählt man beispielsweise eine Breite von 24 Stunden, so werden häufig nur die Maxima täglicher regulärer Schwankungen markiert, die für das Finden von Nachrichten nicht interessant sind.

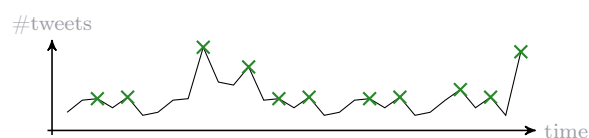


Abbildung 5.9.: Simple Auswahl der lokalen Maxima

Diese einfache Methode kann durch einige Erweiterungen gute Ergebnisse erzielen. Das ist beispielsweise der Fall, wenn eine starke Glättung vorangestellt wird, ein breites Fenster gewählt wird und die zusätzliche Einschränkung gemacht wird, dass Peaks nicht nur das absolute Maximum in ihrem Umfeld sein müssen, sondern auch um einen bestimmten Schwellwert größer sind als ihr Umfeld. Dabei müssen allerdings einige Parameter

festgelegt werden, und da die TPS-Graphen zu verschiedenen Suchbegriffen sehr unterschiedliche Formen haben, war es nicht möglich, allgemein passende Parameter zu finden, die bei vielen Suchbegriffen gute Resultate erzielt haben. Bei folgenden Versuchen wurde entsprechend versucht, eine möglichst geringe Anzahl von Parameter zu verwenden.

Ein alternativer Ansatz ist es, starke Abweichungen von einer Ausgleichskurve zu markieren. Dazu wird zuerst eine Regressionsfunktion bestimmt und dann von zusammenhängenden Teilstücken des Ursprungsgraphen, die über dieser Regressionsfunktion liegen, das jeweilige Maximum als Peak markiert. Da viele Parameter und starke Spezialisierung zu vermeiden waren, wurden statt einer bestimmten Regressionsfunktion für unsere Anwendung eine konstante Funktion gewählt, die so liegt, dass unter ihr genau der Anteil  $\alpha$  der kleinsten Werte des Graphen liegen. Dieses  $\alpha$  ist standardmäßig auf 80% gesetzt und der einzige Parameter.

Um durch den Tages- und Wochenverlauf bedingte zyklische Schwankungen aus der Betrachtung herauszunehmen, wurde in Betracht gezogen, mit einer simplen Fouriertransformation eine Funktion zu bestimmen, die den Normalverlauf beschreibt, sodass dieser Wert von der Kurve abgezogen werden kann. Zum einen spricht gegen dieses Vorgehen, dass sich dieser Normalverlauf in seiner Amplitude verändern kann, durch stetige Veränderungen in der Relevanz eines bestimmten Themas. Zum anderen gibt es Suchbegriffe, deren relevante Peaks auch zyklisch auftreten, wie zum Beispiel bei der regelmäßig ausgestrahlten Fernsehserie Tatort. Stattdessen wurde die Lösung gewählt, zunächst tageweise Maxima zu bestimmen, dann über den diesen Werten die Peaks zu bestimmen und die so identifizierten Peaks dann wieder auf den TPS-Graph zu legen.

Der vollständige Prozess zum Identifizieren von Peaks besteht nun aus 3 Schritten und ist in Abbildung 5.10 illustriert. Zunächst werden in Schritt (1) die tageweisen Maxima bestimmt, um Schwankungen über den Tagesverlauf herauszufiltern. Dann wird in einem zweiten Schritt (2) die  $\alpha$  (Standard: 80%) Grenze und die zusammenhängend darüber liegenden Teilstücke erkannt. Dahinter steckt die Überlegung, dass ein Peak mindestens zu den 20% größten Werten gehören muss. In Schritt (3) werden die Maxima dieser zusammenhängenden Teilstücke markiert und auf den Ausgangsgraphen gelegt.

Auf diese Weise werden Ergebnisse erzielt, die dicht an der Intuition der Benutzer liegen. Was mit dieser Methode nicht automatisch erkannt werden kann ist, wenn an einem Tag mehrere getrennte Ereignisse zu verschiedenen Zeitpunkten stattfinden. Falls der Nutzer selbst aber mehrere Peaks an einem Tag erkennen sollte, so steht ihm die Möglichkeit offen,

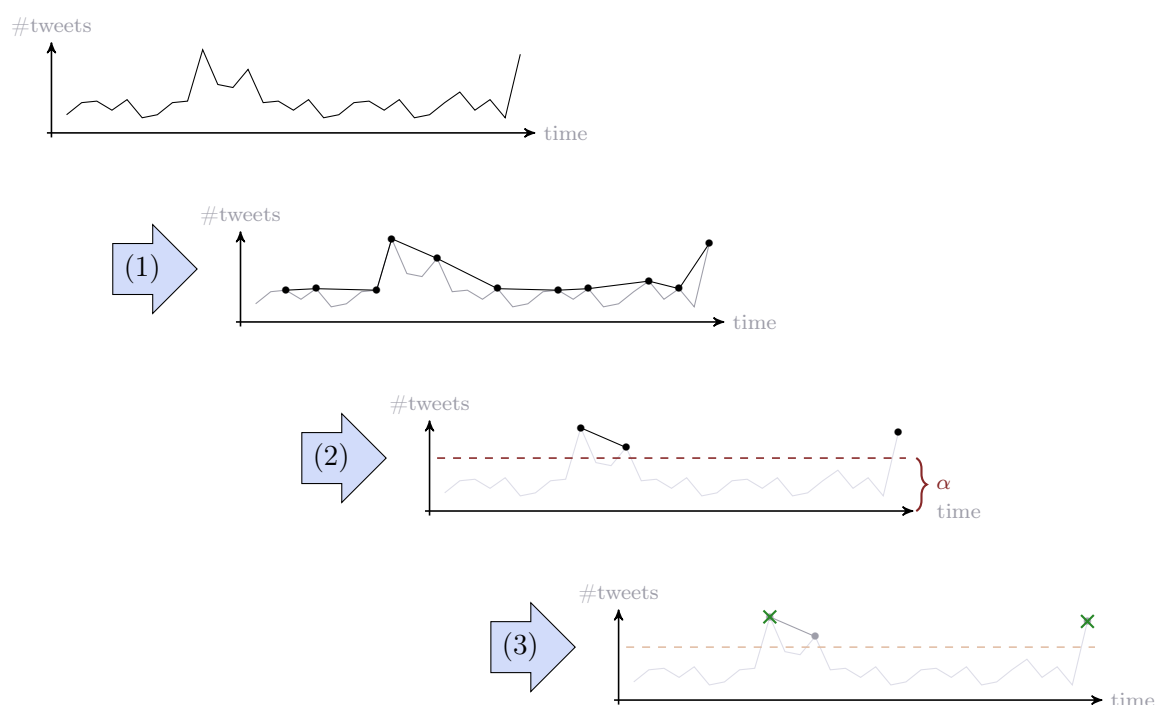


Abbildung 5.10.: Auswahlprozess zur Identifikation von Peaks

durch einen Klick an die entsprechende Stelle der Kurve auch von nicht vorselektierten Zeitpunkten Nachrichten angezeigt zu bekommen.

### 5.3.2. Sammeln passender Nachrichten

Um zu allen gefundenen Peaks oder zu einem vom Benutzer ausgewählten Zeitpunkt Nachrichten für einen bestimmten Suchbegriff zu sammeln, werden Anfragen an die Anbieter Google News<sup>1</sup>, Bing News<sup>2</sup> und Bing Web<sup>3</sup> gestellt. Sie wurden ausgewählt, da sie die Möglichkeit bieten, die Ergebnisse als RSS-Feed auszugeben, und somit ihre Ausgabe leicht verarbeitet werden kann. Google News und Bing News geben in den meisten Fällen allerdings nur Nachrichten zurück, die jünger sind als 30 Tage. Daher wurde auch die Websuche von Bing miteinbezogen.

Über diese Anfragen können spezifisch zu einem Tag und einem Suchbegriff Nachrichten angefordert werden. Gewünscht sind aber nicht nur tagesgenau sondern auch stunden- genau relevante Nachrichten. Dazu werden, wie auch in Abbildung 5.11 illustriert ist,

<sup>1</sup>[news.google.com](http://news.google.com)

<sup>2</sup>[news.bing.com](http://news.bing.com)

<sup>3</sup>[www.bing.com](http://www.bing.com)

parallel Tweets aus der Datenbank passend zum Suchbegriff und der konkreten Stunde gesammelt. Danach werden unter den erhaltenen Nachrichten jene ausgesucht, welche die größte Ähnlichkeit mit den Tweets besitzen. Dabei ist Ähnlichkeit definiert als Ähnlichkeit der Worthistogramme. Die Worthistogramme beschreiben die Anteile, mit denen jedes Wort in einem Text vorkommt und werden mit einer simplen Funktion verglichen, die gemeinsame Worte zählt und häufige Worte stärker gewichtet.

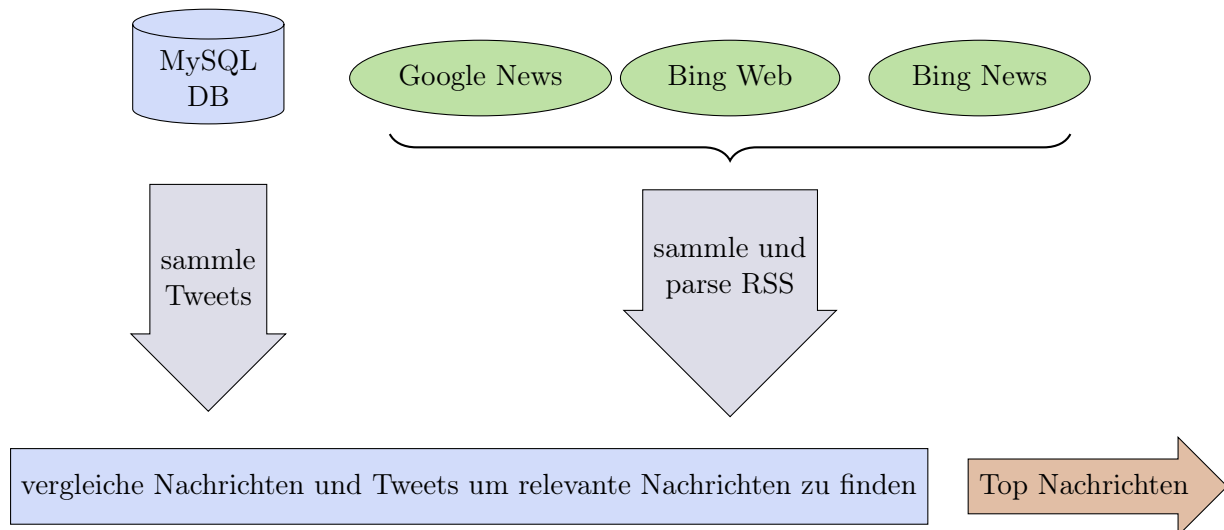


Abbildung 5.11.: Verfahren zum Sammeln von Nachrichten

Werden vom Nutzer durch das Klicken auf einen konkreten Zeitpunkt des TPS Graphen Nachrichten angefragt, ist das Sammeln der Nachrichten je Anbieter und das Sammeln der Tweets in unterschiedlichen Threads parallelisiert. Wenn die Peaks bestimmt wurden und für alle Peaks Nachrichten angefragt werden, dann wird auch dieser Prozess in je einem Thread pro Zeitpunkt parallelisiert, die dann wiederum Threads für das Sammeln der Nachrichten je Anbieter und Sammeln der Tweets starten. Die Anfragen an die Datenbank werden über einen Mutex koordiniert sequentiell gestellt, weil dadurch eine bessere Performanz erzielt wurde. So werden die Anfragen an Nachrichten-Anbieter, die eigene Datenbank und das Verarbeiten ankommender Daten weitestgehend parallelisiert.

Das Modul erfüllt in seiner momentanen Realisierung die Bedürfnisse des Kunden. Die Geschwindigkeit ist hauptsächlich durch die Performanz der Datenbank bei der Abfrage der Tweets begrenzt. Eine Erweiterungsmöglichkeit ist, gefundene Peaks und entsprechende Nachrichten in der Datenbank zu speichern, statt sie bei jeder Anfrage neu zu

bestimmen. So wären alte Nachrichten der Anbieter Google News und Bing News auch über deren Zeitgrenzen hinaus verfügbar sind.

## 5.4. Frontend

### 5.4.1. Design

#### Grober Entwurf

Zu Beginn der Entwicklung des Frontend stand die Frage nach dem Aussehen und dem grundsätzlichen Aufbau im Raum. Es bestand der Wunsch nach einem übersichtlichen und vor allem intuitiv bedienbarem Interface. Erste Überlegungen und Skizzen führten schnell zu der Idee, das Grundlayout an bekannte und erfolgreiche Portale wie Google oder Wolfram Alpha anzulehnen. Die Startseite von TMetrics besteht also lediglich aus einem zentralen Eingabefeld mit darüber angeordnetem Logo und einigen weiteren Einstellungs- und Informationsmöglichkeiten als eine Leiste am oberen Rand.

#### Erstes Konzept: Tabelle

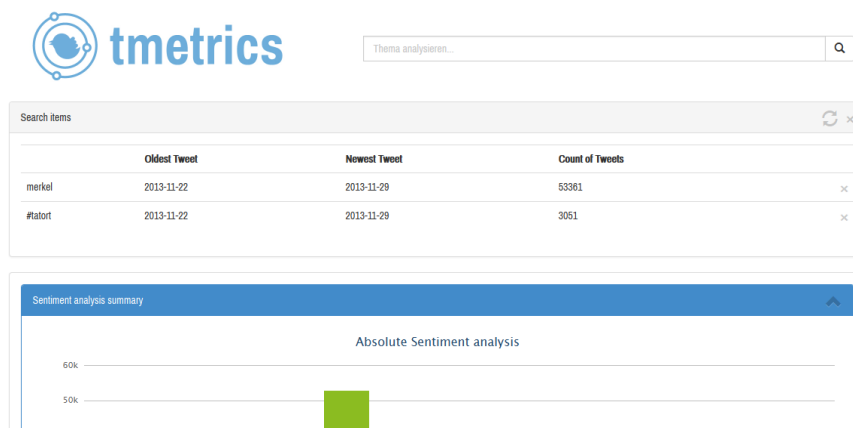


Abbildung 5.12.: Tabellarische Darstellung der Suchbegriffe

Für die weiteren Ansichten wurde ein Konzept erarbeitet, welches diese Einfachheit und Übersichtlichkeit fortführen sollte. Grundsätzlich wurden das Logo und die Suchleiste verkleinert, weiter nach oben gerückt und nebeneinander anstatt untereinander angeordnet. Als dann die ersten Ergebnisse wie die Anzahl der gefundenen Tweets zu einem Suchbegriff, Datum der ersten und der neusten Suche zur Verfügung standen, wurden diese Daten tabellarisch dargestellt. Darunter wurden nach und nach untereinander die einzelnen Auswertungen hinzugefügt (siehe Abbildung 5.12).

Nach der Eingabe eines weiteren neuen Begriffes öffnete sich zunächst immer die Vergleichsansicht. Dies entsprach grundsätzlich der Intention des gesamten Projektes, einzelne Begriffe miteinander vergleichen zu können. Um eine ausführlichere Auswertung zu einem einzelnen Begriff angezeigt zu bekommen, genügte ein Klick auf die jeweilige Tabellenzeile.

Im weiteren Verlauf des Projektes stellte sich doch heraus, dass dieses Bedienkonzept nicht besonders intuitiv zu sein schien. So musste für einen Wechsel zwischen der detaillierten Einzelansicht zweier verschiedener Begriffe mehrfach geklickt werden, was teilweise für den Benutzer nicht direkt nachvollziehbar war.

Außerdem stellte sich heraus, dass mit steigender Anzahl von Analyseansichten der Platz durch die Tabelle direkt im zentralen Fokus des Benutzers eher verschwendet wurde, anstatt diese Fläche mit einer höheren Informationsdichte z.B. in Form einer Analyse zu füllen. Nicht nur der Kunde, sondern auch die Entwickler merkten schnell, dass man hier viel scrollen musste, um die einzelnen Auswertungen dargestellt zu bekommen.

## Zweites Konzept: Kacheln

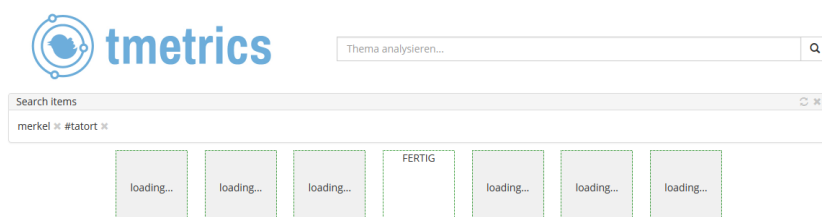


Abbildung 5.13.: Erster Entwurf des Interfaces mit Kacheln, Arbeitsversion

In einem Sprint-Review schlug der Kunde also ein „Kachelinterface“ vor. Der Hintergrund des Kunden dabei war, dass ein „exploratives“ Arbeiten ermöglicht werden sollte, indem man alle Auswertungen als mehr oder weniger große Kacheln direkt zu sehen bekommt und man durch ein Anklicken dieser Kachel in diese Analyseansicht „hineingeht“, man sozusagen auch optisch und informativ in die Tiefe geht und dort dann die Details der jeweiligen Analyse angezeigt bekommt.

Während im Entwicklerteam die ersten Ideen und Ansätze (siehe Abbildung 5.13) zur Umsetzung heranreiften, stellte sich heraus, dass dies an vielen Stellen im Frontend einer völligen Neuentwicklung gleich käme, sich also nur mit erheblichem Aufwand umsetzen ließe. Außerdem fiel anhand erster Skizzen und Überlegungen zur optischen Gestaltung

schnell auf, dass man hier für den eigentlich als essentiell erachteten Vergleich einer bestimmten Analyse zweier verschiedener Begriffe nur noch umständlich zugreifen kann. Grob umschrieben hätte sich der Interaktionsablauf so dargestellt, dass man eine geöffnete Analyse erst hätte schließen müssen, dann den Begriff wechseln um erneut im explorativen Stil wieder in die Tiefe gehen zu müssen.

Aus diesen Gründen beschäftigte sich das Team mit einem alternativen Ansatz.

### **Finales Konzept: Tabs**

Erste Ideen zu diesem Konzept ergaben sich bereits während der Betrachtungen des Kachelinterfaces. Nachdem das Team diese Überlegungen weiter ausführte und gemeinsam mögliche Schwachstellen beseitigte wurde das neue Konzept dem Kunden in einem gesondertem Termin vorgestellt. Dieser war von der Idee begeistert und gab grünes Licht für die Umsetzung. Das Konzept lehnt sich an das Kachelinterface des Kunden an, verbindet aber auch weitere bekannte strukturgebende Elemente wie z.B. verschiedene Tabs. Die Kacheln wurden aufgegriffen, um die Auswahl der zur Verfügung stehenden Analysemöglichkeiten darzustellen. Hier kann der Kunde die gerade aktiven und für alle Suchbegriffe gültigen Analysen einzeln aktivieren oder wieder deaktivieren. Diese Kachelansicht enthält exemplarische Darstellungen und kurze Beschreibungen der jeweiligen Analysen. Für einzelne Suchbegriffe werden nun Reiter anstelle von Tabelleneinträgen genutzt, diese können wie man es von modernen Browsern gewöhnt ist natürlich auch wieder geschlossen werden. Sobald mehr als ein Suchbegriff als Reiter geöffnet ist, erscheint als zusätzlicher Reiter ganz rechts der Comparison-Tab. Hier ist die Darstellung des direkten Vergleichs zweier Begriffe wiederzufinden. Aber auch der Vergleich anderer Analysen zu unterschiedlichen Begriffen ist mit diesem Konzept für den Benutzer deutlich einfacher zu realisieren. So kann er in der Analyseauswahl einfach nur die Analysen aktivieren, die für ihn gerade von Interesse sind. Um zwischen den Analysen der jeweiligen Begriffe hin- und herzuschalten, genügt jetzt ein einziger Klick auf den jeweiligen Tab.

Das ursprüngliche Ziel, ein simples und vor allem intuitives Interface zu gestalten, wurde durch die Verwendung von Reitern und weiteren bekannten optischen Elementen wie z.B. einem geöffnetem bzw. geschlossenem Auge zur Darstellung der aktiven bzw. inaktiven Analysen wieder erreicht.



## 5.4.2. Verwendete Technologien

Bereits vor und während der Phase eines ersten Entwurfs überlegten wir uns, welche externen Technologien und Bibliotheken uns bei der Entwicklung eines Frontends im Browser möglichst gut unterstützen könnten.

Dabei fiel schnell die Entscheidung, das HTML5-Framework Bootstrap [10] einzusetzen. Zwar muss man sich zusätzlich in dieses Framework einarbeiten und HTML5-Frameworks sind generell nicht mit älteren Browsern (wie beispielsweise Internet Explorer 8 oder älter) kompatibel, aber dafür bieten HTML5-Frameworks und im Speziellen Bootstrap auch zahlreiche Vorteile: Mit Hilfe vordefinierter Elemente (Icons, Buttons, Tabs, modale Dialoge, u.v.m.) sowohl in CSS als auch teilweise in JavaScript ist es möglich, ein homogenes Gesamtbild der Oberfläche zu erreichen, ohne selbst viele Anpassungen diesbezüglich machen zu müssen. Außerdem ist Bootstrap zu vielen aktuellen Browsern kompatibel, ohne dass der Entwickler einen Mehraufwand bzgl. der Kompatibilität betreiben muss [11]. Dies verdeutlicht Tabelle 5.6 anhand der Darstellung eines Buttons in verschiedenen Browsern mit und ohne Bootstrap.

Ebenfalls integriert ist der Ansatz von *responsive webdesign*, das heißt, dass sich der Aufbau der Seite an das Endgerät des Benutzers anpassen kann, wofür im Allgemeinen CSS Media Queries [40] eingesetzt werden. Beispielsweise ist es mit CSS Media Queries möglich, die Darstellung von HTML-Elementen an die Breite des Ausgabegerätes anzupassen. Speziell für Bootstrap spricht, dass der Quellcode frei verfügbar ist, viele Menschen es nutzen und somit auch viele Möglichkeiten für Hilfestellungen existieren, sowie die zahlreichen, verfügbaren Beispiele auf der Homepage, die gut dokumentiert sind.









	Firefox	Chrome (Windows)	IE 10	Chrome (Android)
mit Bootstrap				
ohne Bootstrap				

Tabelle 5.6.: Vergleich der Darstellung eines Buttons in verschiedenen Browsern mit und ohne Bootstrap

Mit der Entscheidung für Bootstrap ging auch die Entscheidung für jQuery [37] als Werkzeug zur Navigation und Manipulation des Document Object Models, einem Interface zum dynamischen Zugriff auf die Struktur, den Inhalt und den Stil eines Dokuments, einher. Dies ist dadurch begründet, dass einerseits alle JavaScript-Elemente von Bootstrap jQuery benötigen und wir andererseits aus Erfahrung wussten, dass eine Bibliothek wie jQuery die Gestaltung einer Oberfläche mit Hilfe von JavaScript erheblich vereinfacht.

Zusätzlich zu diesen beiden Bibliotheken, die einen Rahmen zur Entwicklung der Oberfläche bieten, verwenden wir auch für die Darstellung der einzelnen Auswertungen zwei externe Bibliotheken. Dies ist erstens Highcharts [17], welches die Darstellung verschiedener Graphen (beispielsweise Punktwolken, Balken- oder Kuchendiagramme) übernimmt. In der finalen Version unseres Systems wird Highcharts für die Darstellung von sieben der acht verfügbaren Auswertungen verwendet. Für die Darstellung der achten Auswertung (namentlich die Ansicht `tagCloud`) verwenden wir die D3 Word Cloud [14], welche Anordnung, Skalaierung und Färbung der Wörter in der Tag Cloud berechnet und anschließend darstellt.

### 5.4.3. Datenhaltung

Bereits während der ersten Iteration unseres Projekts ergab sich das Bedürfnis, Antworten des REST-Services zwischenspeichern, solange sich der Benutzer auf TMetrics befindet, um möglichst selten neue Anfragen an den REST-Service senden zu müssen. Dies wurde vor allem dann deutlich, wenn bereits mindestens zwei Suchbegriffe eingegeben waren und man zwischen diesen hin- und herwechselte. Bei jedem Wechsel wurden die Anfragen für alle Auswertungen zum Suchbegriff, auf den man wechselte, neu abgeschickt. Die erste Idee, diesem Problem zu entgegnen, war eine Zwischenspeicherung der Daten im Frontend. Da es sich bei den Antworten des REST-Services wie in Abschnitt 3.2 erwähnt um JSON-Objekte handelt, war eine Speicherung dieser ohne großen Zusatzaufwand möglich. Die Frage war nun, wo diese JSON-Objekte abgespeichert werden sollten. Da jedes JSON-Objekt immer genau einem Suchbegriff und einem Suchbegriff mehrere JSON-Objekte (für jede Auswertung ein Objekt) zugeordnet waren, war die erste Idee die in jQuery enthaltene Funktion `data(key, value)` zu nutzen. Diese Funktion speichert zu einem oder mehreren HTML-Elementen ein beliebiges JavaScript-Objekt `value` unter dem Schlüssel `key` ab. Das abgespeicherte Objekt kann mit der Funktion `data(key)` und demselben Schlüssel dann wieder abgerufen werden [38].

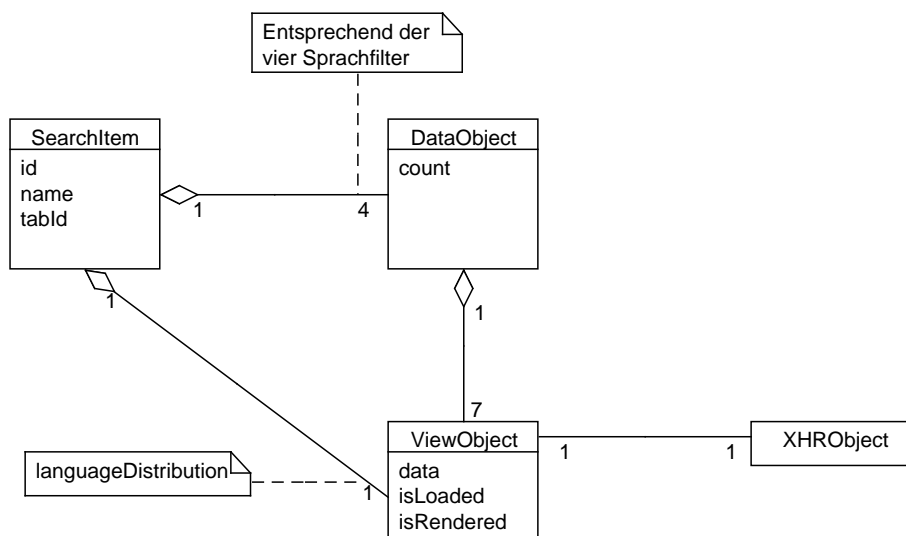


Abbildung 5.14.: Klassendiagramm der Datenhaltung im Frontend

Da in der ersten Version des Frontends zu jedem Suchbegriff auch eine Zeile in der in Abschnitt 5.4.1 beschriebenen Tabelle, also ein `<tr>`-Element, existierte, war die Idee jeder Anfrage einen eindeutigen Schlüsselnamen zu geben und so das entsprechende JSON-Objekt mit diesem Schlüsselnamen an der zu diesem Suchbegriff passenden Tabellenzeile zu speichern. Beim Abrufen einer Auswertung wurde dann, falls verfügbar, auf das zwischengespeicherte JSON-Objekt zugegriffen und ansonsten eine neue Anfrage an den REST-Service gestellt.

Allerdings war diese Umsetzung der Datenhaltung an die Darstellung als Tabelle gebunden, was bei der Umstellung auf ein neues Darstellungskonzept, wie in 5.4.1 beschrieben, zu Problemen führte. Es wäre möglich gewesen, die Daten nun an den neuen HTML-Objekten zu speichern, was das Problem allerdings nur temporär bis zur eventuell nächsten Umstellung des Designs gelöst hätte. Für eine langfristige Lösung war es also von Nöten, Datenhaltung und Darstellung unabhängig voneinander zu gestalten.

Die Umsetzung dieser Idee ist in der in Abbildung 5.14 dargestellten Klassenstruktur eines globalen Arrays von Suchbegriffen gemündet. Dabei wird für jeden Suchbegriff (Klasse `SearchItem`) zwischen vier vordefinierten Sprachfiltern (keine Filterung, englisch, deutsch und eine beliebige weitere Sprache; Klasse `DataObject`) und in jedem Filter nochmals zwischen den acht verfügbaren Ansichten (Klasse `ViewObject`) unterschieden. Die einzige Ausnahme hiervon bildet die Ansicht `languageDistribution`, weil diese unabhän-

gig vom eingestellten Sprachfilter ist und daher nur einmal pro Suchbegriff gespeichert werden muss. Als zusätzliche Information wird zu jedem `DataObject` abgespeichert, wie viele Tweets zu diesem Suchbegriff mit dem jeweiligen Sprachfilter vorliegen. Zu jeder einzelnen Ansicht wird wiederum einerseits, falls vorhanden, das JSON-Objekt der Antwort einer Anfrage des REST-Services zusammen mit den Informationen, ob die Ansicht fertig geladen und gerendert wurde (jeweils als `boolean`-Flag), gespeichert. Andererseits wird die REST-Anfrage selbst als eigenes Objekt `XHRObject` für die Verwendung in den *connection pools* gespeichert, die im nächsten Abschnitt 5.4.4 beschrieben werden. Außerdem sind noch der aktive Tab, der aktive Sprachfilter und ein `boolean`-Array, das angibt, welche Ansichten sichtbar bzw. nicht sichtbar sind, global verfügbar, sodass jederzeit bekannt ist, welche Ansichten der Benutzer aktuell angezeigt bekommt und welche Daten dafür benötigt werden.

Zusätzlich zu der gerade beschriebenen flüchtigen Speicherung von Daten im Frontend, wird auch eine persistente Speicherung für benutzerspezifische Einstellungen verwendet. Dabei werden die Einstellungen zur Sichtbarkeit der Ansichten und der gewählte Sprachfilter über den `localStorage` [41] client-seitig gespeichert. Zudem werden auch die aktuellen Suchbegriffe gespeichert, sodass diese beim nächsten Besuch von TMetrics dem Benutzer in einem Dropdown-Menü zur erneuten Analyse vorgeschlagen werden. Hierbei werden aber immer nur die Suchbegriffe des direkten vorherigen Besuchs gespeichert.

#### 5.4.4. Connection Pools

##### Problematik

Bedingt durch die REST-Architektur unseres Systems und die Entscheidung, einzelne Ansichten unabhängig voneinander aufrufen zu können, wird in unserem System für jede Ansicht (einen Überblick über die Ansichten von TMetrics wurden in Abschnitt 1.2 gegeben und sind in Abbildung 1.1 zu sehen) eine Anfrage an den REST-Service gestellt. Alle gängigen Browser unterstützen aber nur eine gewisse Anzahl an parallelen Verbindungen pro Server (siehe Tabelle 5.7). Das Minimum der Anzahl paralleler Verbindungen zu einem Server unter den von uns unterstützen Browsern liegt bei sechs.

Gibt der Benutzer im Frontend einen Suchbegriff ein und schickt diesen ab, so werden für die Ansichten bereits acht Anfragen an den REST-Service geschickt, was acht parallelen Verbindungen zu einem Server entspricht. Der Browser des Benutzers wird in diesem Fall aber zunächst nur so viele Anfragen abschicken, wie seine Konfiguration es zulässt. Wie

Browser	Parallele Verbindungen
Chrome 4+	6
Firefox 3+	6
IE 8 & 9	6
IE 10	8
Opera 10	8
Opera 11+	6
Safari 4+	6

*Tabelle 5.7.:* Anzahl paralleler Verbindungen zu einem Server verschiedener Browser und Versionen [1].

oben beschrieben gehen wir hierbei im schlechtesten Fall von sechs Anfragen aus. Die verbleibenden anderen beiden Anfragen werden solange zurückgehalten, bis auf eine der vorherigen eine Antwort des REST-Services kommt.

Dies ist soweit noch kein Problem, da der Benutzer lediglich für zwei Auswertungen länger warten muss, als diese Auswertungen eigentlich an Zeit benötigen. Problematisch wird es, wenn der Benutzer direkt nach Eingeben des ersten Suchbegriffs einen weiteren Suchbegriff eingibt, da auch dabei schon bevor die acht Anfragen der Auswertungen abgeschickt werden, bis zu zwei weitere sequentielle Anfragen an den REST-Service geschickt werden. Zuerst die Anfrage, ob der Suchbegriff bereits in der Datenbank vorliegt und falls dies zutrifft die Anfrage, die die interne ID dieses Suchbegriffs zurückliefert. Erst nach der Antwort auf diese beiden initialen Anfragen wird dem Benutzer ein Feedback im Frontend in Form des Erscheinen eines neuen Tabs für den soeben abgeschickten Suchbegriff gegeben und die Anfragen der einzelnen Ansichten werden abgeschickt.

Im ungünstigen Fall, dass nun alle sechs Verbindungen zum Server durch Anfragen der Ansichten belegt sind, wird beim Abschicken eines neuen Suchbegriffs die Anfrage, ob dieser Begriff bereits in der Datenbank vorliegt, zunächst nicht abgeschickt und der Benutzer bekommt somit auch kein Feedback vom Frontend.

Zur Lösung dieser Problematik haben wir zwei Ansätze in Betracht gezogen. Erstens das Zusammenführen von Anfragen, sodass wir insgesamt deutlich weniger Verbindungen pro Server benötigen und zweitens die Einführung eines Verwaltungssystems für Anfragen, sodass wir an Stelle des Browsers regeln, welche Anfragen gestartet und welche noch warten sollen. Das für unsere Entscheidung ausschlaggebende Argument war, dass für den Benutzer häufig nicht alle Ansichten relevant sind. Das heißt, dass er in den Einstellungen zur Sichtbarkeit die Ansichten ausgewählt hat, die er angezeigt bekommen möchte. Unser

System sollte also die Anfragen zu diesen Ansichten bevorzugt behandeln. Dies wäre mit dem ersten Lösungsvorschlag nicht möglich, da alle Anfragen zu einer gebündelt würden und eine Antwort auf diese gebündelte Anfrage erst käme, wenn alle Teilauswertungen vom REST-Service bearbeitet wären. Der Benutzer müsste also im schlimmsten Fall auf die Auswertung einer Ansicht warten, die er nicht sehen möchte. Unser Lösungsansatz ist also die Einführung eines Verwaltungssystems für Anfragen, welches wir *connection pools* nennen.

## Aufbau

Die Idee der *connection pools* ist es, Anfragen nach Prioritäten zu behandeln und abzuarbeiten. Dabei gibt es folgende drei Prioritäten: passiv, aktiv und spezial. Für jede existiert jeweils ein Pool mit momentan in Bearbeitung stehenden Anfragen und eine Warteschlange für noch zu bearbeitende Anfragen. Die Aufgabe der *connection pools* ist es nun, dafür zu sorgen, dass einerseits in den Pools der drei Prioritäten zusammen maximal sechs Verbindungen zum Server existieren und andererseits die Pools mit Anfragen aus den Warteschlangen aufgefüllt werden, wenn eine Anfrage abgearbeitet wurde.

## Umsetzung

Zur Realisierung dieser Aufgaben wird jedem der drei Pools eine feste Anzahl an erlaubten Verbindungen zum Server, sogenannte Slots, zugeteilt. In unserem System stehen für den Spezial-Pool ein, für den Aktiv-Pool drei und für den Passiv-Pool zwei Slots zur Verfügung. Diese Zuteilung ergab sich aus der Überlegung, welche Anfrage mit einer bestimmten Priorität in welcher Häufigkeit auftritt.

Da nur die Anfragen nach der ID eines Suchbegriffs und ob ein Suchbegriff bereits in der Datenbank vorliegt, die Priorität spezial haben und diese Anfragen nie parallel ausgeführt werden, genügt logischerweise ein Slot für den Spezial-Pool. Diese Anfragen sollten also schnellstmöglich abgeschickt werden, um das eingangs genannte Problem zu lösen. Alle anderen Anfragen haben entweder die Priorität aktiv, wenn der Benutzer die Ergebnisse der Anfrage direkt angezeigt bekommen soll oder passiv, falls die Anfrage nur im Hintergrund läuft und Ergebnisse dann auf Abruf bereit stehen sollen. Dies ist abhängig vom aktiven Tab, dem ausgewählten Sprachfilter, sowie der Einstellung für die Sichtbarkeit der einzelnen Ansichten. Eine andere Verteilung der Slots von Passiv- und Aktiv-Pool ist auch möglich und einfach im Code anzupassen.

Bei den Warteschlangen handelt es sich um Stacks bzw. LIFO-Warteschlangen. Dies hat den Hintergrund dass wir dem Benutzer generell das Ergebnis, auf die von ihm zuletzt ausgeführte Aktion am schnellsten liefern möchten, damit die aktuelle Anzeige im Frontend mit Daten gefüllt ist.

Ein weitere Idee der Umsetzung der *connection pools* ist es, dass auf Eingaben des Benutzers angemessen reagiert wird. Das bedeutet, dass die Prioritäten von Anfragen geändert werden, wenn sich die Sichtbarkeit der zugehörigen Ansichten als Folge einer Nutzereingabe ändert. Dies kann durch Wechsel des Tabs, des Sprachfilters oder der Sichtbarkeitseinstellung geschehen. Eine Anfrage zu einer Ansicht, die nicht mehr sichtbar ist, wird, sofern sie noch nicht fertig abgearbeitet ist, abgebrochen und in den Passiv-Pool bzw. in die Passiv-Warteschlange verschoben. Im umgekehrten Fall wird in den Aktiv-Pool bzw. in die Aktiv-Warteschlange verschoben. So wird sichergestellt, dass die Prioritäten aller Anfragen zu jeder Zeit an die Aktionen des Nutzers angepasst sind und Anfragen relevanter Ansichten möglichst schnell bearbeitet werden.

## Fazit

Ein Problem der *connection pools* ist es, dass damit nur Anfragen verwaltet werden, die wir auch aktiv zuweisen. Dies ist bei Anfragen, die über externe Bibliotheken abgeschickt werden, nicht ohne Weiteres möglich. Dies betrifft in unserem System die Autovervollständigung bei Eingabe eines Suchbegriffs. In diesem Fall wird in der Bibliothek `typeahead.js` [16] geregelt, wann genau eine Anfrage abgeschickt wird und wir können sie daher nicht in die *connection pools* einpflegen. Um dieses Problem zu lösen, wäre eine Anpassung der externen Bibliothek nötig.

Ein weiteres mit den *connection pools* verbundenes Problem ist die teilweise schlechte Auslastung der einzelnen Pools. Da eine Anfrage eine festgelegte Priorität hat, ist sie auch fest einem Pool, bzw. einer Warteschlange zugeordnet. Will der Benutzer aber beispielsweise die Auswertungen zu allen Ansichten haben, so werden acht Anfragen an die Connection Pools gesendet, von denen aber nur drei direkt abgeschickt werden, da der Aktiv-Pool nur drei Slots hat. Ohne die *connection pools* wären an dieser Stelle sechs Anfragen abgeschickt worden. Eine mögliche Lösung wäre die Nutzung von Pools anderer Prioritäten unter der Bedingung, dass diese entweder frei sind oder eine niedrigere Priorität haben. So würden Anfragen der Priorität aktiv nur den Spezial-Pool benutzen, wenn dieser frei wäre (und von möglicherweise ankommenden Anfragen mit Priorität spezial wieder verdrängt werden). Der Passiv-Pool hingegen könnte direkt von Anfragen mit

Priorität aktiv genutzt werden. Mögliche Anfragen der Priorität passiv sollten dabei von denen der Priorität aktiv verdrängt werden.

Zusammenfassend lässt sich sagen, dass die *connection pools* eine funktionierende Umsetzung für eine Verwaltung von Anfragen an den Server sind. Um die Effizienz voll auszureizen, sollten aber die in diesem Unterabschnitt genannten Probleme noch behoben werden.

### 5.4.5. Besonderheiten

In diesem Abschnitt werden weitere interessante Details der Implementierung oder Designentscheidungen, die wir bezüglich des Frontends getroffen haben, diskutiert.

#### Anzahl an Tweets im zeitlichen Verlauf

Die im Folgenden beschriebene Ansicht gibt an, wie viele Tweets zu einem gegebenen Suchbegriff in einem bestimmten Zeitraum veröffentlicht wurden.

Bei der ersten Implementierung dieser Ansicht haben wir festgelegt, dass die Tweets eines Tages jeweils addiert werden, sodass der Graph die Anzahl Tweets pro Tag widerspiegelte. Wie in Abbildung 5.15 deutlich wird, wird die Auswertung vor allem bei Suchbegriffen, zu denen nur über einen verhältnismäßig kurzen Zeitraum Daten vorliegen, aussagekräftiger, wenn man den betrachteten Zeitraum von einem Tag auf eine Stunde verkürzt.

Aber auch hierbei ist eine weitere Optimierung denkbar, da die Anzahl Tweets pro Stunde an Aussagekraft verlieren, wenn man einen Datenbestand betrachtet, der sich über einen großen Zeitraum, beispielsweise mehrere Monate, erstreckt. Eine Anpassung des zusammenzufassenden Zeitraums in Abhängigkeit vom gesamten betrachteten Zeitraum wäre also wünschenswert, um diese Problematik zu lösen. Eine leichte Umsetzung dieser Idee ist allerdings nicht möglich, da Highcharts diese Funktionalität nicht nativ unterstützt und man dies manuell einbauen müsste. Im Verlauf des Projektes wurde allerdings weitere Funktionalität in die hier beschriebene Ansicht eingefügt. Nachdem der Kundenwunsch nach explorativem Arbeiten in unserem System laut wurde, wurde einerseits das durch Klick auf den Graphen aufrufbare Anzeigen von einzelnen Tweets einer bestimmten Stunde (siehe Abschnitt 5.4.5) implementiert. Andererseits wurde dann auch noch das in Abschnitt 5.3 beschriebene News-Modul in diese Ansicht integriert. Details dazu sind den jeweiligen Abschnitten zu entnehmen.





Abbildung 5.15.: Vergleich der Tweets im zeitlichen Verlauf pro Tag (links) und pro Stunde (rechts) zum Suchbegriff „Snowden“

## Tweets anzeigen

Wie bereits im vorigen Abschnitt erwähnt, bietet TMetrics die Möglichkeit, explorativ zu arbeiten und sich einzelne Tweets anzeigen zu lassen. Diese Funktionalität ist einerseits in der Auswertung der Sentiment-Analyse verfügbar und andererseits in der Ansicht der Anzahl Tweets im zeitlichen Verlauf. Aus der Sentiment-Analyse heraus werden je nachdem, welcher Teil des Balken- oder Kuchendiagramms angeklickt wurde, nur positive, nur negative oder nur neutrale Tweets angezeigt. Will man einzelne Tweets aus der Ansicht des zeitlichen Verlaufs heraus betrachten, so werden nur Tweets innerhalb des ausgewählten Zeitraums, also innerhalb einer bestimmten Stunde, angezeigt.

Unabhängig von der Ansicht, ist die Anzeige einzelner Tweets auf maximal 100 Tweets begrenzt. Eine solche Begrenzung ist sinnvoll, da sonst bei einem Suchbegriff mit großem Datenbestand sehr lange Ladezeiten verursacht werden können. Um dem Nutzer nun dennoch möglichst repräsentative Tweets zeigen zu können, werden die Tweets nach Wichtigkeit sortiert. Als Maß für die Wichtigkeit eines Tweets sind verschiedene Möglichkeiten denkbar. Wir haben uns dafür entschieden, die Anzahl Retweets eines Tweets als alleiniges Maß für die Wichtigkeit eines Tweets zu nehmen. Grund für diese Entscheidung ist, dass die Anzahl Retweets ein eigenes Feld der Datenbank ist und wir daher effizient sortieren können, um die 100 wichtigsten Tweets zu finden. Da aber auch einem Retweet selbst dieselbe Anzahl an Retweets wie dem Original-Tweet zugeordnet ist, hat dies zur Folge, dass wir Retweets von der Anzeige ausschließen müssen, um eine mehrfache Anzeige des gleichen Tweets zu vermeiden. Es werden also die 100 Original-Tweets mit der höchsten Anzahl an Retweets angezeigt.

Ein weiteres Feature der Anzeige einzelner Tweets ist die Anzeige von weiteren Informationen zu diesem Tweet, die auf Wunsch abrufbar sind. So ist es durch einen einfachen

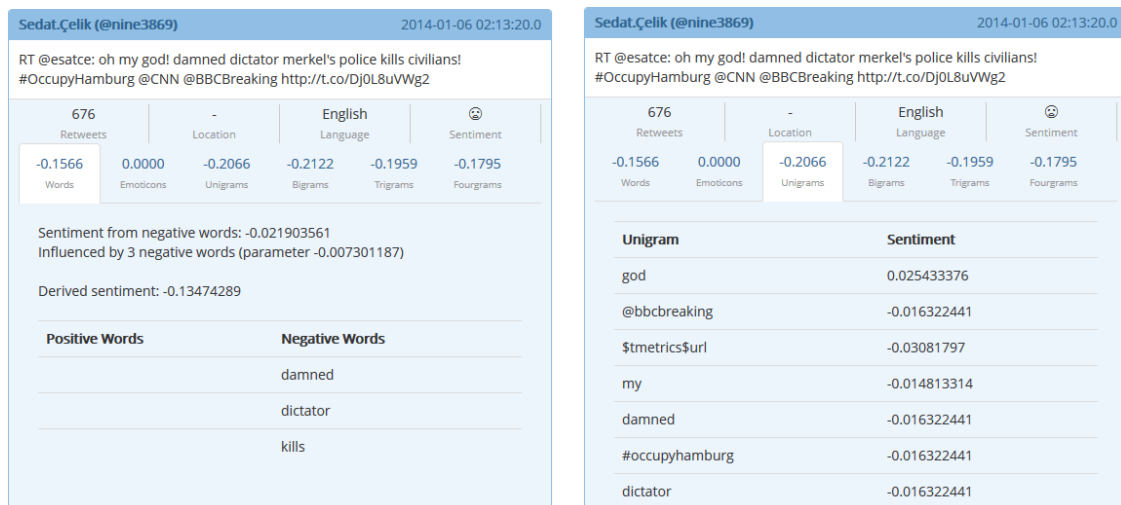
Klick auf den Namen des Autors eines Tweets möglich, nähere Informationen über diesen einblenden zu lassen. Selbiges gilt auch für einen Klick auf den Text des Tweets, was eine Anzeige der Anzahl Retweets, der geografischen Koordinaten (falls verfügbar), der Sprache, sowie des Sentiments des Tweets zur Folge hat. Hier wird dem Benutzer wiederum die Möglichkeit gegeben, sich näher über die Einflussfaktoren des Sentiments zu informieren, was im folgenden Abschnitt 5.4.5 näher erläutert wird. Bei der Designentscheidung war es besonders wichtig, den Fokus zunächst auf das Wesentliche – also den Tweet selbst – zu legen und trotzdem die Möglichkeit zu bieten, detaillierte Informationen dem Nutzer auf Abruf zur Verfügung zu stellen.

### **Darstellung der Sentiment-Einflussfaktoren**

Aus dem Anspruch, dem Benutzer ein exploratives Arbeiten auf TMetrics zu ermöglichen, ergab sich im Projektverlauf die Anforderung, eine Möglichkeit zu liefern, die Einflussfaktoren bei der Bestimmung des Sentiments eines Tweets darzustellen. Grundsätzlich existieren dabei zwei Kategorien von Einflussfaktoren auf das Sentiment eines Tweets: ein vorbestimmtes Wörterbuch (für Wörter und Emoticons) mit festgelegten Sentiment-Werten sowie ein auf Trainingsdaten basierendes Regressionsmodell mit Werten für sämtliche Wörter (Unigrams) sowie Wortgruppen aus zwei bis vier Wörtern (Bigrams, Trigrams und Fourgrams, siehe auch Abschnitt 5.1.2). Daraus ergeben sich insgesamt sechs Faktoren, deren Summe das Sentiment des Tweets bestimmt: Wörter, Emoticons, Unigrams, Bigrams, Trigrams und Fourgrams.

Da für jeden Faktor auch noch angegeben werden soll, wie die Wörter im Tweet den Einfluss dieses Faktors beeinflussen, muss eine Menge Informationen wiedergegeben werden. Zur gleichen Zeit sollte die Größe der Box zum Anzeigen eines Tweets möglichst gering sowie die Darstellung der und Navigation zwischen den Einflussfaktoren möglichst intuitiv gehalten werden. Die Umsetzung dieser Anforderungen ist im Anschluss beschrieben.

Es wurde eine zusätzliche Ansicht hinzugefügt, die ausgehend von der Leiste mit den Metadaten durch Klick auf die Sentiment-Sparte ausgeklappt werden kann. Diese besteht dann aus sechs Tabs, welche den sechs genannten Einflussfaktoren entsprechen. Der Name des Reiters beinhaltet den Wert des Einflussfaktors, sodass die Summe der Werte aller Reiter dem Gesamtsentiment entspricht. Ein Klick auf den Reiter stellt eine detaillierte Auflistung dar, wie sein Wert aus dem Text des Tweets zustande kommt. Diese unterscheidet sich für die zwei eingangs genannten Kategorien.



(a) Wörterbuch (b) n-Gramm

Abbildung 5.16.: Darstellung der Einflussfaktoren des Sentiments

Für Wörterbuch-Faktoren (in Abbildung 5.16a exemplarisch für Wörter) basiert der Wert nur auf der Anzahl gefundener negativer bzw. positiver Wörter/Emoticons und einem zugehörigen Parameter. Ist mindestens ein solches Wort vorhanden, werden diese Werte und daraus berechnete (positive oder negative) Teil-Sentiment dargestellt. Der dabei dargestellte abgeleitete Sentiment-Wert („derived sentiment“) entspricht der Aggregationsfunktion zum Wörterbuch von Liu (siehe Abschnitt 5.1.2). Dieser wird immer angezeigt. Abschließend werden sämtliche im Wörterbuch vorkommenden Wörter des Tweets gelistet.

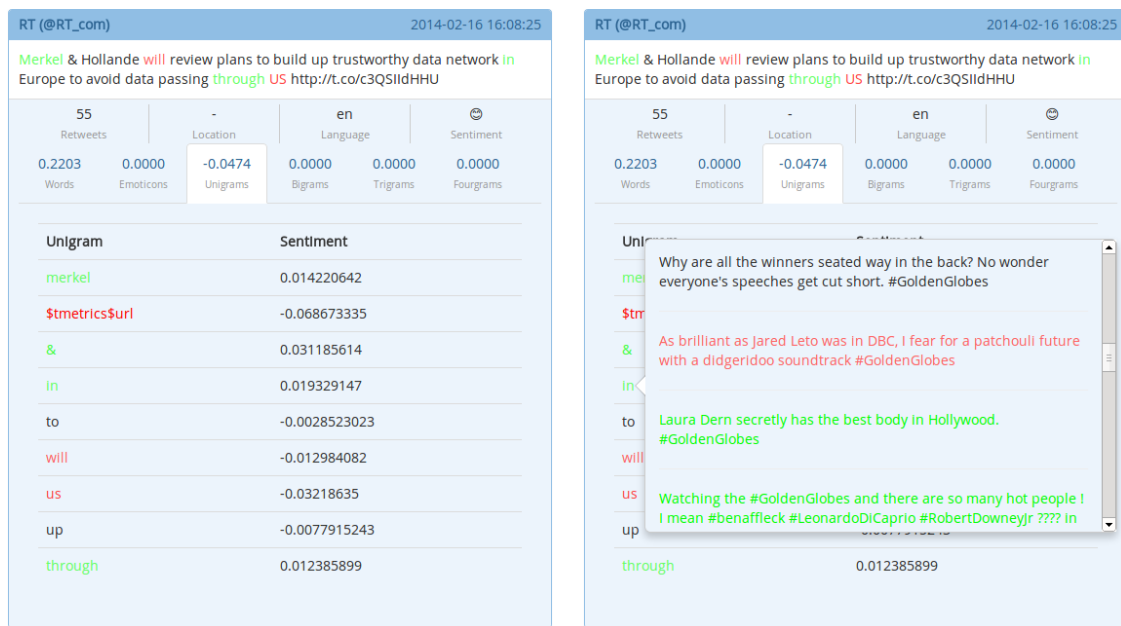
Die Darstellung der Details von Unigrams (für Unigrams in Abbildung 5.16b analog für Bigrams, Trigrams, Fourgrams) besteht nur aus einer tabellarischen Auflistung dieser n-Gramms des Tweets und der dazugehörigen Sentimentwerte (der Gesamtwert des Einflussfaktors entspricht der Summe seiner n-Gramms).

Zur besseren visuellen Aufbereitung dieser Informationen wird außerdem noch eine Einfärbung der n-Grams nach ihrem Sentiment-Wert vorgenommen. Dabei wurde die Konvention verwendet, neutrales Sentiment wie gewöhnlich in Schwarz darzustellen, während positives und negatives Sentiment in vier grünen bzw. roten Farbtönen mit zunehmender Sättigung dargestellt wird. Die dabei notwendige Festlegung wurde nach Betrachtung des Wertebereichs des Sentiments einzelner n-Grams intuitiv vorgenommen. Für positive n-Grams wurde festgestellt, dass nur sehr positive n-Grams den Wert 0.05 überschreiten, weshalb dies als untere Grenze für den grünen Farbton mit der höchsten Sättigung ver-

wendet wurde. Weiterhin haben wir es als sinnvoll erachtet, dass nur dann ein positives Sentiment vorliegt, wenn die zweite Nachkommastelle nicht null ist. Daher wurde 0.1 als obere Grenze des neutralen Sentiments und damit als Untergrenze des grünen Farbtönen mit der niedrigsten Sättigung verwendet. Danach fiel die Wahl auf 0.25 und 0.4 als Grenzwerte zwischen den übrigen Farbtönen. Da das Sentiment vom Modell her symmetrisch ist, wurden dieselben Grenzen im Negativen für die roten Farbtöne des negativen Sentiments verwendet. Da im Wörterbuch alle positiven Einträge gleich positiv und alle negativen Einträge gleich negativ sind, haben wir dort von einer Einfärbung abgesehen, da sie keine weiteren Informationen liefert.

Nachdem auf diese Weise bereits eine Heuristik zur Einfärbung von n-Grams implementiert worden ist, wurde die Entscheidung getroffen, sie auch auf den Gesamttext des Tweets anzuwenden. Eine Einfärbung auf Grundlage sämtlicher n-Grams ist dabei nicht praktikabel, da nicht für alle Wörter im Text ein eindeutiges Sentiment existiert. Es könnte z.B. ein Bigram mit negativem Sentiment existieren, das ein Unigram mit positivem Sentiment beinhaltet. Außerdem ist das Einfärben nur im Fall der Unigrams trivial, da sich dort der Text leicht in seine Wortbestandteile zerlegen lässt und sich diese anhand des Sentimentwertes des entsprechenden Unigrams einfärben lassen. Aus diesem Grund wurde das Einfärben nur für Unigrams umgesetzt. Die ursprüngliche Zielsetzung, die Einfärbung vom offenen Tab in der Ansicht der Einflussfaktoren abhängig zu machen und bei geschlossener Detailansicht gar keine Einfärbung im Volltext vorzunehmen, konnte aus Zeitgründen nicht umgesetzt werden, ist aber grundsätzlich möglich und sinnvoll. Unter diesen Umständen wäre es außerdem auch möglich, eine Einfärbung des Volltexts auf Grundlage des Wörterbuchs vorzunehmen. Beide Arten der Einfärbung sind in Abbildung 5.17a zu sehen.

Bei Betrachtung der Einflussfaktoren ist es möglich, dass die Fragestellung auftritt, wie das Sentiment dieses Faktors zustande gekommen ist (beispielsweise bei einem negativen Wert für ein intuitiv positives Wort). Aus diesem Grunde kann es von Interesse sein, sich die Trainingstweets anzuschauen, die zu dieser Bewertung des Sentiments geführt haben. Da unter Umständen viele Trainingstweets zu einem n-Gram vorliegen und jedes Mal deren Volltext dargestellt werden muss, stellt sich wieder die Herausforderung, diese Menge an zusätzlichen Informationen intuitiv und ohne Verkomplizierung des bestehenden Interfaces darzustellen. Dies wurde schlussendlich durch die Möglichkeit gewährleistet, auf ein n-Gram zu klicken und damit ein scrollbares Popover zu öffnen, welches nur den Text der relevanten Trainingstweets auflistet. Da Trainingstweets nur mit den Werten -1, -0.5,



(a) Einfärbung: n-Grams und Volltext

(b) Trainingstweets als Popover

Abbildung 5.17.

0, 0.5 und 1 gelabelt werden können, ist es außerdem nicht notwendig, diese Werte explizit aufzulisten. Stattdessen wird das Labeling der Trainingstweets durch entsprechendes Einfärben des gesamten Textes in entsprechenden Farbtönen analog zur Einfärbung der n-Grams dargestellt. Ein Beispiel für die Darstellung der Trainingstweets kann man in Abbildung 5.17b sehen.

## Sentiment im zeitlichen Verlauf

Schon sehr früh im Verlaufe des Projektes war es möglich, den Verlauf des Meinungsbildes zu einem konkreten Suchbegriff darzustellen. Hierzu wurde der anteilige Verlauf der positiven und der negativen Tweets mittels zweier sich überlagernder Kurven dargestellt. (Siehe Abbildung 5.18.) Obwohl diese Darstellung zunächst vielversprechend zu sein schien, stellte sich im weiteren Verlauf heraus, dass dieser Graph oft fehlinterpretiert wurde.

Die Kenntnis über die absolute Summe der jeweils positiven und negativen Tweets zu einem Zeitpunkt ist nur von bedingtem informativen Nutzen. Ausserdem unterliegen die absoluten Werte der Tweets zu jedem Zeitpunkt durchaus starken Schwankungen, so dass es schwierig für den menschlichen Beobachter ist, aus dieser Art der Darstellung

Informationen über die zeitliche Verteilung zwischen positiven und negativen Tweets herauszufiltern.

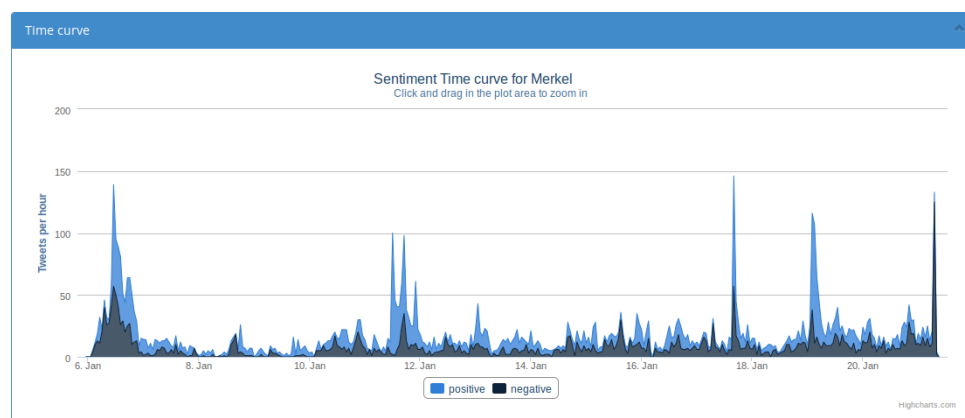


Abbildung 5.18.: Vorherige Version der Sentimentdarstellung im zeitlichen Verlauf.

Daher wurde eine abweichende Darstellung entwickelt, welche das relative Verhältnis zwischen positiven und negativen Tweets darstellt. Dies ermöglicht dem Benutzer deutlich besser zu erkennen, wie sich das Meinungsbild zu dem jeweiligen Suchbegriff über den zeitlichen Verlauf entwickelt hat.

Aber auch bei dieser Darstellung gab es zwei verschiedene Probleme. Zum einen gab es unabhängig von der zeitlichen Auflösung immer einige Bereiche in denen es keine Tweets gab, weder positive noch negative, sodass viele Lücken in dem Graphen vorhanden waren, was es schwerer machte, diesen zu interpretieren. Zum anderen gab es ebenso Zeitpunkte, in denen nur positive oder nur negative Tweets gefunden werden konnten, sodass auch hierdurch die Darstellung stark zerhackt wurde. Diese Probleme wurden durch den Einsatz eines gleitenden Mittelwertes behoben. Der Graph zeigt somit zu jedem Zeitpunkt den Durchschnitt des Verhältnisses zwischen positiven und negativen Tweets der vorherigen 24 Stunden an.

Diese Darstellung wurde von allen Beteiligten direkt verstanden und kann intuitiv auch ohne weitere Erklärungen von jedem Benutzer erfasst werden. (Siehe Abbildung 5.19.)

## Performance der Tag Cloud

Die Darstellung der Tag Cloud funktioniert wie gewünscht, es konnten jedoch Ladeprobleme bei den anderen Analysen beobachtet werden, seit die Tag Cloud zu den Auswertungen hinzugefügt wurde. Teilweise stockte die gesamte Darstellung der Auswertungen. Erste

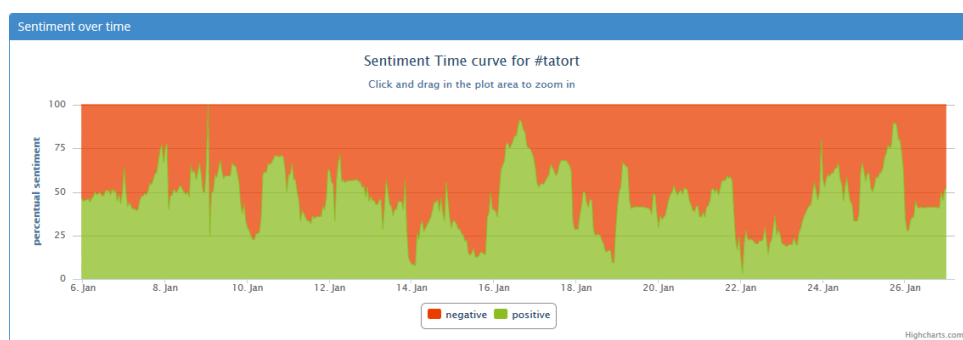


Abbildung 5.19.: Finale Version der Sentimentdarstellung im zeitlichen Verlauf.

Überlegungen in Richtung Datenübertragung oder Verzögerungen auf Serverseite wurden schnell zerstreut, da andere Auswertungen hier länger auf Daten warten müssen und die Darstellung anderer Auswertungen hierdurch nicht verzögert wird.

Die weitere Analyse beschäftigte sich also mit dem Aufbau der eingesetzten Fremdkomponenten. Dies ergab, dass die `d3-cloud`-Bibliothek für jeden Eintrag einer Liste von Wörtern versucht, eine mögliche Position zu ermitteln. Diese Positionsermittlung geschieht in zentrischen Spiralen und benötigt dazu exakte Informationen über die Ausdehnung des jeweiligen Wortes, um auch die Lücken zwischen einzelnen unterschiedlich hohen Buchstaben des selben Wortes ausnutzen zu können. Versuche ergaben, dass sich unabhängig von der Länge dieser Liste die Anzahl der anzeigbaren Worte circa auf 100 belief. Obwohl der verfügbare Platz zur Anzeige ab dann nahezu vollständig aufgebraucht war, wurden genannte Berechnungen für jedes weitere Wort in der Liste vorgenommen. Diese Berechnungen konnten mittels kürzerer Listen als die Ursache für die beobachteten Probleme identifiziert werden. Obwohl mittels JavaScript-Timer- und Intervall-Funktionen versucht wurde für diese Berechnungen eine gewisse Asynchronität zu erreichen, sodass diese Berechnungen nicht mehr zwingend am Stück durchlaufen müssen, sondern von dem JavaScript-Interpreter unterbrochen werden können, um zunächst die anderen Auswertungen darzustellen, war dieser Ansatz von unzureichendem Erfolg gekrönt. Die Begrenzung der Liste schien das adäquate Mittel der Wahl zu sein, zusammen mit den Optimierungen in Punkto asynchroner Ausführung konnte somit eine zufriedenstellende Performance-Optimierung erreicht werden, ohne durch eine zu große Beschränkung der Liste Nachteile im Detailgrad der Darstellung in Kauf nehmen zu müssen.

Zwischenzeitlich kam die Idee auf, die gesamte Berechnung evtl. auf den performanteren Server zu verlagern. Da die Darstellung der Tag Cloud aber von dem verfügbarem Platz

auf der Webseite und vor allem auch von der verwendeten Schriftart abhängig ist, lässt sich diese Berechnung in zufriedenstellendem Maße nur im Browser bewerkstelligen. Andernfalls hätte eine komplette Neuentwicklung der Tag Cloud stattfinden müssen, welche im Ergebnis fertige Grafiken zum Browser übertragen hätte und nicht wie es aktuell der Fall ist, lediglich eine schmale Liste von Wörtern im JSON-Format. Dieser Ansatz wurde daher auch sehr schnell wieder verworfen.

## **Sprachfilter**

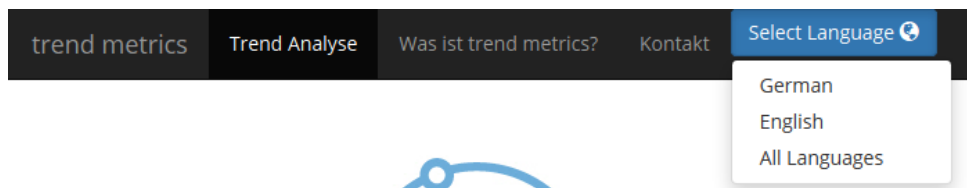
Die Idee der Einbindung eines Sprachfilters entstand, nachdem die Ansicht der Sprachverteilung implementiert war und der Kunde den Wunsch hegte, einzelne Sprachen eines Suchbegriffs genauer analysieren zu können. Der Sprachfilter ist eine globale Einstellung und wirkt sich auf die Auswertungen aller Suchbegriffe aus, die der Benutzer angezeigt bekommt.

Auf dieser Basis musste nun entschieden werden, wo in der Anzeige des Sprachfilters am besten einzubauen ist. In der ersten Version wurde ein einfaches Dropdown-Menü in der Navigationsleiste der Seite verwendet. Wie in Abbildung 5.20a zu sehen ist, war dieser Ansatz aber funktional getrieben, da die Anzeige des Sprachfilters nicht in das optische Gesamtkonzept integriert ist. Im zweiten Konzept (siehe 5.20b) fügt sich der Sprachfilter nun optisch ins Gesamtbild ein. Die Anbringung an die Suchleiste soll dem Benutzer suggerieren, dass es sich um eine Filterfunktion handelt und nicht um eine Einstellung der Anzeigesprache der Seite. Ein Problem ist allerdings, dass eine Änderung des Sprachfilters nur möglich ist, wenn die Suchleiste auch sichtbar ist. Da die Suchleiste aber verschwindet, wenn der Benutzer bei der Analyse der Ansichten nach unten scrollt, bedeutet dies, dass der Benutzer zunächst wieder nach oben scrollen muss, bevor der Sprachfilter geändert werden kann. Vor allem bei mobilen Endgeräten tritt dieses Problem aufgrund der geringeren Höhe der Ausgabe häufig auf. Aus dieser Überlegung entstand in der Folge das finale Konzept (siehe 5.20c), bei dem eine homogene Einbindung in die Navigationsleiste vorliegt. Daher ist der Sprachfilter auch immer sichtbar und verfügbar, da die Navigationsleiste beim Scrollen immer sichtbar bleibt.

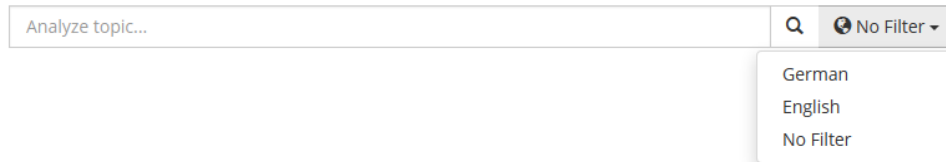
## **Warnungen und Fehler**

Damit mögliche Fehler als Antwort einer Anfrage vom REST-Service nicht zum Absturz des gesamten Frontends führen, ist es notwendig, angemessen auf Fehler zu reagieren.

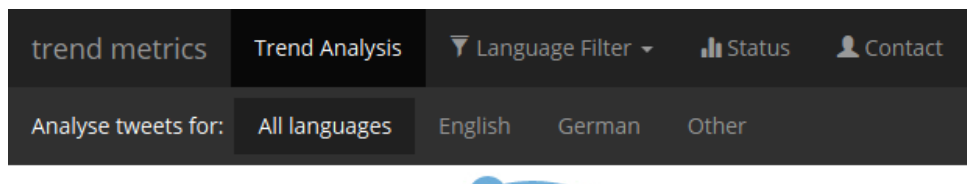




(a) Erste Version des Sprachfilters. Ansatz: Funktionalität statt Design.



(b) Zweite Version des Sprachfilters. Ansatz: Sprachfilter an der Suchleiste.



(c) Finale Version des Sprachfilters. Ansatz: Integration in die Navigationsleiste

Abbildung 5.20.: Vergleich der verschiedenen Layouts des Sprachfilters

Wenn der HTTP Status Code der Antwort nicht 200 OK lautet, handelt es sich um einen Fehler, bzw. ist ein Fehler im REST-Service aufgetreten. Eine Warnung liegt vor, wenn die Antwort vom REST-Service fehlerfrei vorliegt, aber nicht genügend Daten vorhanden sind, um vernünftige Auswertungen anzeigen zu können.

Die Anzeige der Fehler und Warnungen erfolgte zunächst in einer separaten Box, die aber nur die Anzeige eines einzigen Fehlers ermöglichte. Diese Einschränkung wurde getroffen, damit der Benutzer nicht von einer Flut an Fehlermeldungen erdrückt wird, falls die Verbindung zum REST-Service nicht fehlerfrei abläuft und somit jede Anfrage fehlschlägt.

Da aber Fehler und Warnungen immer mit einer Anfrage an den REST-Service verknüpft sind, erschien es im Laufe der Entwicklung vernünftig, dies auch optisch durch die Anzeige von Fehlern und Warnungen in den einzelnen Auswertungsbboxen hervorzuheben. Fehlermeldungen, die nicht mit einer einzelnen Auswertung zusammenhängen, werden weiterhin in einer separaten Box angezeigt. Allerdings können diese Meldungen nun, ähn-

lich wie ein Fenster einer Anwendung in Windows, in der rechten oberen Ecke geschlossen werden, sodass der Benutzer selbst entscheiden kann, wann eine Meldung für ihn nicht mehr relevant ist.

Zusätzlich dazu wurden auch noch Fehlercodes eingeführt, die in den Fehlermeldungen angezeigt werden. Diese stellen einen Kompromiss zwischen möglichst exakter Fehlerbeschreibung für den Entwickler und Vermeidung unnötiger Details für den Benutzer dar.

### 5.4.6. Diskussion

Die Entwicklung des Frontend hat gezeigt, dass man diesen Teil nicht vollständig von der darunter liegenden Schicht loslösen kann und dass die Absprachen in unserem Fall mit dem REST-Service weit über eine Verständigung über die zu verwendende API hinausgehen müssen. So muss man bei der Konzeption frühzeitig einige mögliche Problemstellen berücksichtigen, wie z.B. die auf im Schnitt maximal sechs mögliche parallele Verbindungen begrenzte Kapazität der Browser-Server-Verbindungen. Man sollte seine Anwendung also auf keinen Fall so bauen, dass man auf viele parallele Verbindungen angewiesen ist, und falls sich dies nicht vermeiden lässt, dass diese auf jeden Fall sehr schnell terminieren um weiterer Verbindungsversuche des Browsers nicht zu blockieren. Dies würde den Browser ansonsten schnell einfrieren oder der Benutzer erhält den Eindruck, dass seine Anwendung gerade hängen geblieben ist. Diese Probleme lassen sich durch frühzeitige Konzepte wie z.B. einer Art Multiplexing umgehen, welche hier nicht weiter ausgeführt werden sollen.

Des Weiteren haben wir erkannt, welche Berechnungen man browserseitig anstellen sollte und welche Berechnungen auf dem Server besser aufgehoben sind. Die Tag Cloud ist hier ein sehr schönes Beispiel. Aufgrund der Vielzahl unterschiedlicher Systeme, die auf einen Webservice zugreifen, kann man auf dem Server keinerlei Annahmen über die beim Besucher vorhandenen Schriftarten machen, welche browserseitig dynamisch für die Anzeige der Wörter verwendet werden. Somit lassen sich die exakten Positionen nicht auf dem Server ermitteln und diese Berechnung muss im Browser stattfinden. Dennoch sollte man sich an dieser Stelle stets der eingeschränkten Möglichkeiten von JavaScript bewusst sein, da zumindest zum heutigen Datum noch keine parallelen Ausführungen von JavaScript möglich sind und zu umfangreiche Berechnungen den Browser sehr schnell blockieren können. Eine sinnvolle Abwägung ist also unabdingbar.

Durch die verschiedenen optischen Ansätze des Frontends ist ebenfalls deutlich geworden, dass auch hier eine detaillierte konzeptionelle Vorarbeit notwendig ist, um spätere weiträumige Änderungen auf ein Minimum zu reduzieren. Weitreichende Änderungen bis hin zu kompletten Neuentwicklungen sind im Verlaufe eines Projektes oftmals sehr schwierig und dann nur mit einem erheblichen Mehraufwand zu bewerkstelligen. Dennoch hat der Verlauf des Projektes hier auch gezeigt, dass man selten im Voraus alle Anforderungen exakt kennt und gerade diese Komponente durch eine starke Nähe zum Kunden flexibel reagieren können muss. Frühzeitige Überlegungen gemeinsam mit dem Kunden, wie das Frontend aussehen soll und eventuell die Anfertigung eines funktionslosen Prototypen können mögliche Wege sein, unnötige Änderungen im späteren Verlauf zu vermeiden. Dennoch kann man festhalten, dass die ändernden Anforderungen zusammen mit dem flexiblen Unterbau in Form des HTML5-Frameworks Bootstrap [10] zu einem sehr zufriedenstellenden Ergebnis geführt haben. Trotz aller Änderungen ist es uns daher gelungen, die gewünschten Anforderungen umzusetzen und sogar zu übertreffen.

## 6. | Ausblick und Fazit

TMetrics ist ein System zur Datenanalyse von zu Suchbegriffen zugehörigen Tweets. Neben dem Sammeln der Tweets ermöglicht TMetrics es dem Benutzer, sich die einzelnen Tweets anzeigen zu lassen und ihren Sentimentwert festzustellen. Ebenfalls wird dem Benutzer die Möglichkeit geboten, sich Nachrichten eines Tages zu einem Begriff anzeigen zu lassen. Zudem gibt es eine Anzeige der häufigsten Begriffe aller Tweets zu einem Suchbegriff ebenso wie die Anzeige der am häufigsten verwendeten Hashtags des Suchbegriffs. Des Weiteren existiert noch die Möglichkeit, sich Cluster von Tweets anzeigen zu lassen.

Das gesamte System besteht aus einem Daemon, einem REST-Service und einem Frontend. Der Daemon sucht permanent nach neuen Tweets zu vorgegebenen Suchbegriffen und speichert diese in einer lokalen Datenbank mitsamt berechnetem Sentiment ab. Der REST-Service beantwortet Anfragen, die vom Frontend aus durch den Benutzer gestellt werden, während das Frontend nur für die Darstellung der anzuzeigenden Daten zuständig ist.

Die Teilnehmer des Projektseminars sind mit dem Gesamtprodukt mit Ausnahme der mangelnden Performance zufrieden. Das zu Beginn des Projektseminars angestrebte Ziel, ein System zur Datenanalyse von Tweets zu schaffen, wurde erreicht. Zwar sind nicht alle geplanten Funktionalitäten wie eine Heatmap zu Suchbegriffen oder ein Kinomodul umgesetzt worden. Aber dafür ist das Nachrichtenmodul als neue, zuvor nicht geplante Funktionalität hinzugekommen.

Die mangelnde Performance, die aufgrund der immer größer werdenden Datenmenge in der Datenbank auftritt, hat sich leider durch das gesamte Projektseminar hinweg gezogen. Es wurden viele Anstrengungen unternommen, diesen Mangel zu beheben. Dazu zählen

- das Anpassen des Datenbankschemas,
- die Einführung von Indizes auf Tabellen innerhalb der Datenbank,
- die Optimierung von SQL-Queries,

- und ein effizienteres Arbeiten mit der Datenbank innerhalb des REST-Services und des Daemons.

Zwar haben all diese Änderungen teilweise eine enorme Verbesserung der Performance relativ zu vorher bewirkt, leider wurde aber dennoch nicht die von uns gewünschte Performance erreicht.

Retrospektiv stellte sich uns die Frage, ob die Entscheidung der Wahl einer relationalen Datenbank, in unserem Fall MySQL, nicht falsch gewesen ist, da in [21, S. 24] eine deutlich höhere Datenbankgeschwindigkeit mit einer NoSQL-Datenbank wie beispielsweise MongoDB [7] erreichbar zu sein scheint. Gerade beim Thema Big Data sollen NoSQL-Datenbanken gegenüber relationalen Datenbanken im Vorteil sein ([35], zitiert nach [21, S. 23]). Vielleicht hätte aber auch ein anderes Datenbankschema bereits die angesprochene Performance radikal verbessert.

Zwar ist die Performance nicht zufriedenstellend, dafür aber funktionieren die einzelnen Komponenten wie Clustering, Sentimentanalyse und Nachrichtenmodul im gewünschten Umfang. Mögliche Optimierungen oder Probleme wurden bereits in den jeweiligen Abschnitten der genannten Komponenten erwähnt.

Trotzdem besteht weiterhin noch großes Ausbaupotential, was allerdings nicht enttäuschend, sondern vielmehr der abgedeckten Breite des Systems geschuldet ist. Diese ermöglichte es nicht auch noch, die einzelnen Aspekte des Systems intensiver zu erweitern oder optimieren. Erweiterungsmöglichkeiten oder mögliche Optimierungsmaßnahmen wurden bereits in den Abschnitten über die einzelnen Aspekte des Systems behandelt.

Der Aufbau des Projektseminars durch Verwendung von Scrum als Entwicklungsmodell hat uns sehr geholfen, iterativ ein funktionierendes System zu schaffen, das mit jeder neuen Iteration erweitert und verbessert wurde. Der wöchentliche Wechsel des Scrum-Masters hat jedem Teilnehmer zeitweise eine Verantwortung über das Projekt gegeben. Somit ist jeder Einzelne in dieser Rolle dazu motiviert, sich mit dem Gesamtsystem auseinander zu setzen und mit den einzelnen Arbeitsbereichen zu kommunizieren, um gegebenenfalls auftretende Schwierigkeiten und Probleme früh zu erkennen. Des Weiteren war das Planning Poker zum Einschätzen des Zeitaufwands hilfreich, um einen realistischen Ziel am Ende einer Iteration anzustreben und zu erreichen. Die Verwendung eines Scrum-Boards ermöglichte es jedem Teilnehmer, immer zu sehen, wie weit das Projekt innerhalb der Iteration bereits fortgeschritten und an welchen Stellen gegebenenfalls noch weitere Hilfe nötig war.

Die Stimmung unter allen Teilnehmern war immer gut, sodass ein harmonisches und motivierendes Arbeitsklima entstand, das die Produktivität gesteigert hat. Insgesamt haben während des Projektseminars alle Teilnehmer viel Neues gelernt, sei es über das Entwickeln mithilfe von Scrum, die Versionierungskontrolle durch Git, besondere Tricks in JavaScript, mögliche Stolpersteine bei der Multi-Threading-Entwicklung, die Verwendung eines Build-Management-Tools wie Maven, das Einrichten eines Servers und vieles mehr, was in Hinblick auf zukünftige (berufliche) Projekte hilfreich sein wird.

# Literaturverzeichnis

- [1] *BrowserScope - Custom browsers network comparison*. <http://www.browserscope.org/?category=network&v=1&ua=Chrome%204%2CChrome%2035%2CFirefox%203%2CFirefox%2028%2CIE%208%2CIE%209%2CIE%2010%2COpera%2010%2COpera%2011%2CSafari%204%2CSafari%207>, [Online; zugegriffen am 20.03.2014].
- [2] *Google Trends, Suchbegriff: walker*. <https://www.google.com/trends/explore/#q=walker>, [Online; zugegriffen am 24.03.2014].
- [3] *Jackson Java-Bibliothek*. <http://wiki.fasterxml.com/JacksonHome/>, [Online; zugegriffen am 26.03.2014].
- [4] *Java Database Connectivity (JDBC) Java-Bibliothek*. <http://www.oracle.com/technetwork/java/javase/jdbc/index.html>, [Online; zugegriffen am 26.03.2014].
- [5] *Jersey Java-Bibliothek*. <https://jersey.java.net/>, [Online; zugegriffen am 26.03.2014].
- [6] *Log4J Java-Bibliothek*. <http://logging.apache.org/log4j/2.x/>, [Online; zugegriffen am 26.03.2014].
- [7] *MongoDB*. <https://www.mongodb.org/>, [Online; zugegriffen am 26.03.2014].
- [8] *Twitter4J*. <http://twitter4j.org/en/index.html>, [Online; zugegriffen am 27.03.2014].
- [9] ANTENUCCI, D., G. HANDY, A. MODI und M. TINKERHESS: *Classification of Tweets via Clustering of Hashtags*, 2011.
- [10] BOOTSTRAP TEAM: *Bootstrap*. <http://www.getbootstrap.com/>, [Online; zugegriffen am 20.03.2014].

- [11] BOOTSTRAP TEAM: *Bootstrap - Browser and device support*. <http://www.getbootstrap.com/getting-started/#support>, [Online; zugegriffen am 20.03.2014].
- [12] BORG, I. und P. J. F. GROENEN: *Modern Multidimensional Scaling: Theory and Applications*. Springer, 2005.
- [13] DATA-DRIVEN DOCUMENTS: *Data-Driven Documents*. <http://www.d3js.org/>.
- [14] DAVIES, J.: *D3 Word Cloud*. <http://www.jasondavies.com/wordcloud/>, [Online; zugegriffen am 20.03.2014].
- [15] FIELDING, R. T.: *Architectural Styles and the Design of Network-based Software Architectures*. Doktorarbeit, University of California, Irvine, 2000.
- [16] HARDING, J., V. SKARICH und T. TRUEMAN: *typeahead.js*. <http://twitter.github.io/typeahead.js/>, [Online; zugegriffen am 20.03.2014].
- [17] HIGHCHARTS AS: *Highcharts*. <http://www.highcharts.com/>, [Online; zugegriffen am 20.03.2014].
- [18] JOLLIFFE, I. T.: *Principal Component Analysis*. Springer Series in Statistics. Springer, 2002.
- [19] KAPPELHOFF, P.: *Multidimensionale Skalierung - Beispiel zur Datenanalyse*, 2001. <http://kappelhoff.wiwi.uni-wuppertal.de/fileadmin/kappelhoff/Downloads/Vorlesung/mds.pdf>, [Online; zugegriffen am 07.02.2014].
- [20] KRUSKAL, J. B.: *On the shortest spanning subtree of a graph and the traveling salesman problem*. Proceedings of the American Mathematical Society, 7(1):48 – 50, 1956.
- [21] KUMAR, S., F. MORSTATTER und H. LIU: *Twitter Data Analytics*. Springer, New York, NY, USA, 2013.
- [22] LIU, B.: *Liu Opinion Lexicon*. <http://www.cs.uic.edu/~liub/FBS/sentiment-analysis.html#lexicon>, [Online; zugegriffen am 26.03.2014].
- [23] LLOYD, S.: *Least squares quantization in PCM*. Information Theory, IEEE Transactions on, 28(2):129–137, Mar 1982.



- [24] OFFIZIELLE MYSQL DOKUMENTATION: *10.1.10.6 The utf8mb4 Character Set (4-Byte UTF-8 Unicode Encoding)*. <https://dev.mysql.com/doc/refman/5.5/en/charset-unicode-utf8mb4.html> [Online; zugegriffen am 27.03.2013].
- [25] OFFIZIELLE TWITTER API DOKUMENTATION: *GET search/tweets*. <https://dev.twitter.com/docs/api/1.1/get/search/tweets/>, [Online; zugegriffen am 24.03.2014].
- [26] OFFIZIELLE TWITTER API DOKUMENTATION: *OAuth*. <https://dev.twitter.com/docs/auth/oauth/>, [Online; zugegriffen am 24.03.2014].
- [27] OFFIZIELLE TWITTER API DOKUMENTATION: *Obtaining access tokens*. <https://dev.twitter.com/docs/auth/obtaining-access-tokens/>, [Online; zugegriffen am 24.03.2014].
- [28] OFFIZIELLE TWITTER API DOKUMENTATION: *REST API Rate Limiting in v1.1*. <https://dev.twitter.com/docs/rate-limiting/1.1/>, [Online; zugegriffen am 24.03.2014].
- [29] OFFIZIELLE TWITTER API DOKUMENTATION: *REST API v1.1 Limits per window by resource*. <https://dev.twitter.com/docs/rate-limiting/1.1/limits/> [Online; zugegriffen am 24.03.2014].
- [30] OFFIZIELLE TWITTER API DOKUMENTATION: *The Streaming APIs*. <https://dev.twitter.com/docs/api/streaming/>, [Online; zugegriffen am 24.03.2014].
- [31] OFFIZIELLE TWITTER API DOKUMENTATION: *Tweets*. <https://dev.twitter.com/docs/platform-objects/tweets/>, [Online; zugegriffen am 24.03.2014].
- [32] OFFIZIELLE TWITTER API DOKUMENTATION: *Users*. <https://dev.twitter.com/docs/platform-objects/users/>, [Online; zugegriffen am 24.03.2014].
- [33] OFFIZIELLE TWITTER API DOKUMENTATION: *Using the Twitter Search API*. <https://dev.twitter.com/docs/using-search/>, [Online; zugegriffen am 24.03.2014].
- [34] PANG, BO und LILLIAN LEE: *Opinion Mining and Sentiment Analysis*. Foundations and Trends in Information Retrieval, 2(1-2):1–135, 2008.

- [35] REDMOND, E. und J. R. WILSON: *Seven Databases in Seven Weeks*. Pragmatic Bookshelf, 2012.
- [36] STIEGLITZ, S. und L. DANG-XUAN: *Social Media and Political Communication — A Social Media Analytics Framework*. Social Network Analysis and Mining, (August 2012), 2012.
- [37] THE JQUERY FOUNDATION: *jQuery*. <http://www.jquery.com/>, [Online; zugegriffen am 20.03.2014].
- [38] THE JQUERY FOUNDATION: *jQuery - API Documentation: data*. <http://api.jquery.com/data/>, [Online; zugegriffen am 20.03.2014].
- [39] TSUR, O., A. LITTMAN und A. RAPPOPORT. In: *Proceedings of the 7th International Conference on Weblogs and Social Media*, Seiten 621–630, 2013.
- [40] WORLD WIDE WEB CONSORTIUM: *Media Queries*. <http://www.w3.org/TR/css3-mediaqueries/>, [Online; zugegriffen am 24.03.2014].
- [41] WORLD WIDE WEB CONSORTIUM: *Web Storage*. <http://www.w3.org/TR/webstorage/>, [Online; zugegriffen am 24.03.2014].

# A. | REST API

## Query-Methoden

POST /queries/post

Parameter	Beschreibung	
q	Suchbegriff	verpflichtend

Legt einen noch nicht existierenden Suchbegriff in der Datenbank an.

POST /queries/postPriority

Parameter	Beschreibung	
id	ID eines vorhandenen Suchbegriffs	verpflichtend
p	Neue Priorität für den angegebenen Suchbegriff	verpflichtend

Ändert die User-Priorität für einen Suchbegriff.

POST /queries/postActiveFlag

Parameter	Beschreibung	
id	ID eines vorhandenen Suchbegriffs	verpflichtend
active	True oder false, je nachdem ob der Suchbegriff für den Daemon aktiv oder inaktiv sein soll (verpflichtend)	verpflichtend

Aktiviert oder deaktiviert einen Suchbegriff für den Daemon.

### GET /queries/byid

Parameter	Beschreibung	
id	ID eines Suchbegriffs	verpflichtend

Liefert den durch die ID spezifizierten Suchbegriff aus der Datenbank zurück, falls vorhanden.

### GET /queries/bystring

Parameter	Beschreibung	
q	Suchbegriff	verpflichtend

Liefert den angegebenen Suchbegriff aus der Datenbank (inklusive ID) zurück, falls vorhanden.

```
{
  stacktrace: null,
  - data: {
    id: 2,
    String: "merkel"
  },
  error_codes: null
}
```

### GET /queries/contains

Parameter	Beschreibung	
q	Suchbegriff	verpflichtend

Liefert zurück, ob der angegebene Suchbegriff in der Datenbank vorhanden ist.

### GET /queries/suggestions

Parameter	Beschreibung	
q	Beginn eines Suchbegriffs	verpflichtend

Liefert eine Liste der Suchbegriffe in der Datenbank zurück, welche mit q beginnen. Die Liste ist absteigend nach Anzahl Vorkommnisse sortiert. Es werden maximal fünf Ergebnisse zurückgeliefert.

#### GET /queries/metadata

Parameter	Beschreibung	
id	ID eines Suchbegriffs	verpflichtend
lang	Sprache der einbezogenen Tweets im ISO-Format	optional

Liefert die Metadaten des Suchbegriffs dieser `id` zurück:

```
{
  stacktrace: null,
  - data: {
    - query: {
      id: 2,
      String: "merkel"
    },
    language: null,
    count: 263600,
    oldest_tweet: "2014-01-05",
    newest_tweet: "2014-02-24"
  },
  error_codes: null
}
```

#### GET /queries/hasDaemonFetched

Parameter	Beschreibung	
id	ID eines Suchbegriffs	verpflichtend

Überprüft, ob der Daemon für den Suchbegriff mit der angegebenen `id`, Tweets gefunden hat.

## Result-Methoden

**GET** /results/tweet

Parameter	Beschreibung	
id	ID eines Tweets	verpflichtend

Liefert alle Informationen für den Tweet dieser `id` aus der Datenbank.

**GET** /results/tweets

Parameter	Beschreibung	
id	ID eines Suchbegriffs	verpflichtend
sent	Sentiment: liefert nur Tweets mit diesem Sentiment zurück, erlaubte Eingaben: „positive“, „neutral“, „negative“ (alle Tweets falls nicht angegeben)	optional
start	Datum im Format YYYY-DD-MMTHH:MM:SS, Datum des ältesten zurückgegebenen Tweets (keine untere Grenze falls nicht angegeben)	optional
end	Datum im Format YYYY-DD-MMTHH:MM:SS, Datum des neuesten zurückgegebenen Tweets (keine obere Grenze falls nicht angegeben)	optional
lang	Sprache der Tweets im ISO-Format wie von Twitter erkannt (alle Tweets falls nicht angegeben)	optional
limit	Anzahl zurückgelieferter Tweets (100 falls nicht angegeben)	optional

Liefert die Tweets, die dem Suchbegriff dieser `id` zugeordnet sind, unter den angegebenen Einschränkungen.

```
{
  stacktrace: null,
  - data: [
    - {
      - tweet: {
        id: "420149270717550593",
        text: "Angela Merkel, the German chancellor, has been injured in a skiing accident
        http://t.co/ZedbkK3QQb",
        - lang: {
          iso_code: "en",
          string: "en"
        },
        - sentiment: {
          value: -0.0476632,
          string: "neutral"
        },
        - sentimentFeatures: {
          words: null,
          emoticons: null,
          + unigrams: [...],
          + bigrams: [...],
          + trigrams: [...],
          + fourgrams: [...],
          + others: [...],
          wordDictionaryDetails: null,
          emoticonDictionaryDetails: null
        },
        coordinate_longitude: 0,
        coordinate_latitude: 0,
        created_at: "2014-01-06 11:06:06.0",
        retweet_count: 136
      },
      - user: {
        id: "19706851",
        name: "Telegraph World News",
        screen_name: "TelegraphWorld"
      }
    },
    ...
  ],
  error_codes: null
}
```

GET /results/user

Parameter	Beschreibung	
id	ID eines Twitter-Users	verpflichtend

Liefert alle Informationen für den User dieser id aus der Datenbank.

### GET /results/sentimentPerHour

Parameter	Beschreibung	
id	ID eines Suchbegriffs	verpflichtend
lang	Sprache der angezeigten Tweets (alle Tweets falls nicht angegeben)	optional

Liefert für jede volle Stunde die Anzahl positiver und negativer Posts zum Suchbegriff mit dieser id zurück.

### GET /results/countPerHour

Parameter	Beschreibung	
id	ID eines Suchbegriffs	verpflichtend
lang	Sprache der angezeigten Tweets (alle Tweets falls nicht angegeben)	optional

Liefert für jede volle Stunde die Anzahl Posts zum Suchbegriff mit dieser id zurück.

```
{
  "stacktrace": null,
  "data": {
    "graph": [
      {
        "count": 110,
        "peak": false,
        "news": [ ],
        "date": "2014-01-05T23:00:00.000"
      }
    ],
    "search_term": {
      "id": 2,
      "String": "merkel"
    }
  },
  "error_codes": null
}
```



## GET /results/tagCloud

Parameter	Beschreibung	
id	ID eines Suchbegriffs	verpflichtend
lang	Sprache der angezeigten Tweets (alle Tweets falls nicht angegeben)	optional
count	Anzahl einbezogener Tweets (100 falls nicht angegeben)	optional

Liefert den konkatenierten Text der festgelegten Anzahl Tweets zum Suchbegriff der angegebenen id zurück, sodass damit auf Client-Seite die Tag-Cloud erstellt werden kann.

## GET /results/sentiments

Parameter	Beschreibung	
id	ID eines Suchbegriffs	verpflichtend
lang	Sprache der angezeigten Tweets (alle Tweets falls nicht angegeben)	optional

Durchsucht die Tweets zum Suchbegriff mit der angegebenen id und liefert zurück, wieviele von ihnen jeweils das Sentiment „positiv“, „neutral“ und „negativ“ haben.

```
{
  stacktrace: null,
  - data: {
    id: 2,
    positive: 13568,
    neutral: 48397,
    negative: 33817
  },
  error_codes: null
}
```

## GET /results/getDataGroups

Parameter	Beschreibung	
id	ID eines Suchbegriffs	verpflichtend
lang	Sprache der angezeigten Tweets (alle Tweets falls nicht angegeben)	optional

Durchsucht die Tweets zum Suchbegriff mit der angegebenen id und liefert zurück, welcher Tweet zu welchem Cluster gehört.

### GET /results/peaks

Parameter	Beschreibung	
id	ID eines Suchbegriffs	verpflichtend
lang	Sprache der angezeigten Tweets (alle Tweets falls nicht angegeben)	optional

Liefert die Peaks des Suchbegriffs dieser id zurück.

### GET /results/news

Parameter	Beschreibung	
id	ID eines Suchbegriffs	verpflichtend
lang	Sprache der angezeigten Tweets (alle Tweets falls nicht angegeben)	optional
day	Tag für den die News abzufragen sind	optional
month	Monat für den die News abzufragen sind	optional
year	Jahrg für den die News abzufragen sind	optional

Liefert die News des Suchbegriffs dieser id für den angegebenen Zeitraum zurück:

```
{
  stacktrace: null,
  - data: {
    - news: [
      - {
        provider: "Bing Web",
        title: "Angela Merkel recovering from skiing accident, says German government 6/1/2014 ",
        url: "http://www.youtube.com/watch?v=Ukis5EJARyk",
        rating: 1.159912037276566,
        text: "Angela Merkel recovering from skiing accident, says German government 6/1/2014"
      },
      - {
        provider: "Bing Web",
        title: "Merkel muss liegen - Aktuelle Nachrichten | RP ONLINE",
        url: "http://www.rp-online.de/video/aktuelles/merkel-muss-liegen-vid-1.3926439",
        rating: 0.7646357416524034,
        text: "Merkel muss liegen; Mein RP ONLINE Benutzernamen / E-Mail-Adresse. Passwort.
              wiederholen. ... 6.1.2014. Merkel muss ..."
      }
    ]
  },
  error_codes: null
}
```

## GET /results/languages

Parameter	Beschreibung	
id	ID eines Suchbegriffs	verpflichtend

Gibt eine Liste von Sprachen zurück (eine Sprache besteht aus isoCode und Häufigkeit).

```
{
  stacktrace: null,
  - data: [
    - {
      count: 48380,
      iso_code: "de"
    },
    - {
      count: 47441,
      iso_code: "en"
    },
    ...
  ],
  error_codes: null
}
```

## GET /results/trainingTweets

Parameter	Beschreibung	
feature	Betrachtetes Feature	verpflichtend
lang	Sprache des Modells	optional

Gibt eine Liste von Tweets zurück, die das angegebene Feature für die spezifizierte Sprache am meisten beeinflusst haben.

## GET /results/hashtagstatistics

Parameter	Beschreibung	
id	ID eines Suchbegriffs	verpflichtend
lang	Sprache der angezeigten Tweets (alle Tweets falls nicht angegeben)	optional

Durchsucht die Tweets zum Suchbegriff mit der angegebenen id und liefert zurück, welcher Hashtags darin vorkommen. Die Ausgabe ist sortiert nach den Häufigkeiten.

```
{
  stacktrace: null,
  - data: {
    - counts: [
      29154,
      5181,
      3134,
      2899,
      2814,
      1573,
      1510,
      1506,
      1506,
      1423
    ],
    search_term_id: 2,
    - hashtag_ids: [
      "26",
      "39626",
      "3415",
      "35935",
      "5669",
      "107118",
      "89284",
      "103267",
      "103266",
      "2119"
    ],
    - hashtag_texts: [
      "occupyhamburg",
      "stopgermanpolicevoilence",
      "nsa",
      "jokogegenklaas",
      "direnhamburg",
      "stopgermanpoliceviolence",
      "followback",
      "bundespresse",
      "webnews",
      "groko"
    ]
  },
  error_codes: null
}
```