Was ist los?

Erhebung und Visualisierung von empirischen Daten verschiedener Internet-Zeitungen

Halbjahresarbeit 2016/2107 von Jaro Habiger

Contents

1 Idee

In meiner Freizeit programmiere ich gerne. Ich mag es, Probleme mit Computern oder anderen technischen Geräten zu lösen. Auch gewinne ich hierbei oft neue Erkenntnisse, da man sich beim Programmieren unkonventionell mit den Dingen beschäftigen muss: Man muss mehr in Strukturen denken, muss versuchen einem Computer, der kein Verständnis von Sinn oder Inhalt hat, das Problem nur in logischen Zusammenhängen zu beschreiben. Somit kann mithilfe eines Computerprogramms fast nie eine wirkliche Lösung für ein Problem oder eine Antwort auf eine Frage gefunden werden, allerdings können Computer einem Menschen helfen, eine bestimmte Aufgabe besser, schneller oder einfacher zu erledigen.

Besonders interessant finde ich es immer dann, diese Macht, die wir uns mit der Programmierung von Computern geben können, zu verwenden, wenn wir sie nutzen um neue Einblicke in unsere Umwelt zu erlangen. Auch hier gilt zwar wieder, dass der Computer eigentlich viel dümmer als wir ist, und keine Zusammenhänge begreifen kann, allerdings stumpfe Aufgaben viel schneller und ausdauernder zu erledigen vermag. Diese Kombination aus Eigenschaften ermöglicht es, große Datensätze mit mathematischen Methoden zu analysieren und die Ergebnisse für Menschen ansprechend aufzubereiten. Bei dieser Aufbereitung bietet sich eine optische Darstellung an, da die Augen des Menschen das Sinnesorgan sind, das am schnellsten Daten erfassen kann [spiegelminig]. Es bietet sich also an, empirische Daten grafisch aufzubereiten, sprich Datenvisualisierung zu betreiben.

Relativ schnell war mir also klar, das ich mich mit Datenvisualisierung beschäftigen will. Anfangs war allerdings nicht ganz klar, welchen Datensatz ich auf welche Aspekte hin untersuchen werde. Eines Tages nahm ich dann eine Ausgabe des Spiegels zur Hand und sah auf der ersten Seite eine sogenannte Wortwolke, in der oft verwendete Worte größer gedruckt waren, als weniger oft verwandte. Diese Form der Darstellung fand ich sofort sehr ansprechend. Ich entschied mich also dazu, verschiedene Texte über das aktuelle Weltgeschehen, die ich von verschiedenen Online-Zeitungen herunterladen kann, zu analysieren. Diese Analyse wollte ich von Anfang an nicht unbedingt nur auf die Generierung von Wortwolken beschränken, sondern mit dem Datensatz “herumexperimentieren”, also gucken, welche interessanten Visualisierungen möglich sind.

Also fing ich an und suchte Zeitungen aus.

2 Die Auswahl der Zeitungen

Als das Thema der Halbjahresarbeit feststand musste ich Zeitungen aussuchen, die ich analysieren will. Hierbei orientierte ich mich an einer Studie der Arbeitsgemeinschaft Online Forschung aus dem November 2014. Hierbei ging ich nach der Anzahl der "Unique Users", also quasi der Leser der jeweiligen Nachrichtenseite. Nach dieser Studie [1] sind die 10 Nachrichtenseiten mit den meisten Lesern:

Rang	Nachrichtenseite	Leser (in Milionen)
1	Bild.de	16,91
2	Focus Online	13,65
3	Spiegel Online	11,43
4	Die Welt	9,56
5	Süddeutsche.de	7,41
6	stern.de	6,45
7	Zeit Online	5,78
8	n-tv.de	4,69
9	FAZ.net	4,54
10	N24.de	4,07

Von diesen Online-Zeitungen habe ich nun Bild.de Focus Online, Spiegel Online ausgewählt. Dies sind die 3 Zeitungen mit den meisten Lesern. Zusätzlich habe ich Süddeutsche.de und Zeit Online ausgewählt, da ich diese Zeitung manchmal selber lese. An dieser Auswahl ist unter anderem sehr interessant, dass sie Nachrichtenquellen verschiedener politischer Ausrichtungen und intellektuellen Anspruchsniveaus enthält. Dies ermöglicht es, am Ende auch Vergleiche zwischen den Zeitungen durchzuführen und eigene Hypothesen leicht zu verifizieren.

[1] http://meedia.de/2015/01/29/agof-news-top-50-n24-zahlen-explodieren-auch-manager-magazin-und-taz-mit-riesen-plus/

3 Konzeptioneller Aufbau

Als nächstes galt es, ein System zu konzipieren, welches die Daten der ausgewählten Zeitungen periodisch herunterlädt, vorverarbeitet und die gesammelten Ergebnisse für spätere Analyse abspeichert. Dieser Schritt ist ein sehr wichtiger, der gut bedacht sein muss, da die am Ende durchführbaren Analysen davon abhängen, welche Daten in diesem Schritt erhoben werden. Ich habe mich dazu entschlossen, die Daten, die ich erhebe, in einer Datenbank zu speichern. Auf diese sollten nun kleine Auswertungsprogramme, die verschiedene Visualisierungen generieren, zugreifen. Diese sollten im Webbrowser laufen, da dieser die Programmierung der Visualisierungen vereinfacht und die Visualisierungen sehr portabel macht. Der gesamte Auswertungsprozess sollte jeweils durch diese Visualisierung vorgenommen werden, um einfacheres Experimentieren mit dem Datensatz zu ermöglichen. Um dies möglich zu machen, ist zusätzlich noch ein "Adapter" zwischen Visualisierung und Datenbank notwendig.

Dieses System sieht am Ende aus wie in Abb. 1 gezeigt.

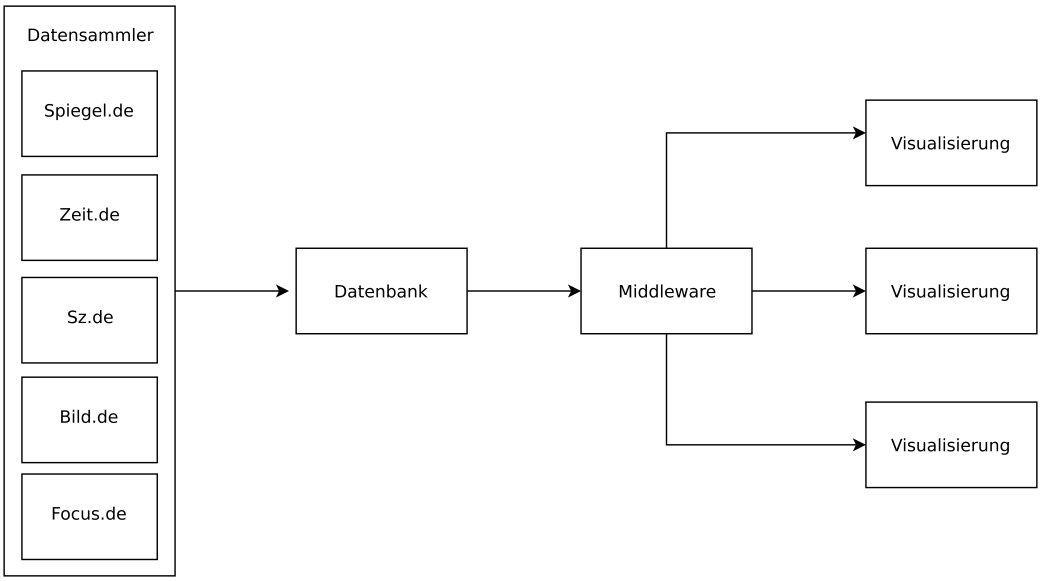

Abb. 1: Der Aufbau des Gesamtsystems

Wie ich dieses System nun umgesetzt habe, ist im nächsten Absatz beschrieben.

4 Umsetzung

Der nun konzipierte Aufbau lässt sich sehr gut in einzelne Teilprobleme unterteilen, was ich bei meiner Umsetzung auch sehr strikt beachtet habe. Hierbei ist es am einfachsten, den Datenfluss von den verschiedenen Nachrichtenquellen zur fertigen Visualisierung zu betrachten. In den nachfolgenden Abschnitten wird dieser Verarbeitungsprozess beschrieben. Die einzelnen Schritte sind teilweise eher schwierig zu verstehen, da ihr Sinn wenig greifbar ist. Dies liegt daran, das viele verschiedene Repäsentationen und Strukturen von Daten ineinander umgewandelt werden müssen.

4.1 Daten sammeln

Als erstes müssen Daten zur weiteren Verwertung von den verschiedenen Nachrichtenquellen gesammelt werden. Dies geschieht über die sogenannten "RSS-Feeds". Bei diesen handelt es sich um ein standardisiertes Format, über das Nachrichtenanbieter ihre Artikel, inklusive Metadaten wie z.B. den Zeitpunkt der Veröffentlichung, in maschinenlesbarer Form bereitstellen. Im Prinzip sind also die gleichen Daten wie auf der normalen Internetseite abrufbar, mit dem Unterschied, dass diese einfacher mit Hilfe von Programmen verarbeitet werden können (Siehe Abb. 2 und 3).

Diese Darstellung als RSS-Feed ermöglicht es, die Artikel verschiedener Online-Zeitungen zu betrachten, ohne für jede Zeitung einen komplett neuen Datensammler programmieren zu müssen.

Der Datensammelprogramm ist eine Software, welche ich in Python geschrieben habe. Python ist eine Programmiersprache, die auf Einfachheit und Flexibilität optimiert ist. Das Datensammelprogramm lädt sich den RSS-Feed einer einzelnen Nachrichtenquelle periodisch herunter und verarbeitet ihn weiter. Ich habe mich dazu entschieden, jeden RSS-Feed einmal pro Minute neu zu analysieren. Im ersten Schritt der Verarbeitung lädt das Programm den Feed als Datei von den Servern der jeweiligen Online-Zeitung herunter. Diese Datei ist eine sogenannte XML-Datei. In ihr werden Daten als Baumstruktur abgebildet. Hierbei muss man sich die Datei als "Stamm" des Baums vorstellen. Die ersten "Äste" der Baumstruktur beinhalten die Metadaten, wie z.B. den Erstellungszeitpunkt und den Herausgeber des

Abb. 2: Ein Artikel einmal in der normalen Darstellung...

```
▼<rss xmlns:content="http://purl.org/rss/1.0/modules/content/" version="2.0">
  ▼<channel>
     <title>SPIEGEL ONLINE - Schlagzeilen</title>
     <link>http://www.spiegel.de</link>
    ▼<description>
       Deutschlands führende Nachrichtenseite. Alles Wichtige aus Politik,
       Wirtschaft, Sport, Kultur, Wissenschaft, Technik und mehr.
     </description>
     <language>de</language>
     <pubDate>Sun, 11 Dec 2016 20:15:15 +0100</pubDate>
     <lastBuildDate>Sun, 11 Dec 2016 20:15:15 +0100</lastBuildDate>
    ▼<image>
       <title>SPIEGEL ONLINE</title>
       <link>http://www.spiegel.de</link>
       <url>http://www.spiegel.de/static/sys/logo_120x61.gif</url>
     </image>
    ▼<item>
      ▼<title>
         Gut zwei Promille: Ostfriese sucht Reeperbahn in Braunschweig
       </title>
      ▼<link>
         http://www.spiegel.de/panorama/gesellschaft/ostfriese-sucht-reeperbahn-in-
         braunschweig-a-1125427.html#ref=rss
       </link>
```

Abb. 3: ... und einmal als Teil eines RSS-Feeds

Feeds. Die nachfolgenden “Äste” enthalten jeweils einen Artikel. Diese Artikel wiederum beinhalten verschiedene Unterelemente, d.h. die “Äste” verzweigen sich in weitere, kleinere “Ästchen”. In diesen stehen nun z.B. Autor, Titel, oder eine Kurzfassung des Textes. Außerdem ist ein Link zur Vollversion des Artikels gegeben. Diese textuelle Repräsentation einer Baumstruktur gilt es nun in eine einfacher verwendbare Repräsentation im Speicher des Python Programms umzuwandeln, also zu “parsen”. Das Wort “parsen” ist sehr schwierig ins Deutsche zu übersetzen. Es beschreibt den Sachverhalt, die textuelle Repräsentation eines strukturierten Datensatzes in die “native” Repräsentation einer Programmiersprache für diesen Datensatz zu bringen. Die “native” Repräsentation eines Datensatzes in einer Programmiersprache ist die Standardform, diesen Datensatz abzubilden. Diese hat oft bestimmte Funktionen, die das Programmieren vereinfachen, wie z.B. die Zugriffsmöglichkeit auf einzelne Elemente bei Listen. Hierfür wird eine Programmbiliothek verwendet, die einen sogenannten “XML-Parser” beinhaltet, der genau dies erreicht. Nach diesem Schritt können die einzelnen Artikel betrachtet werden. Hierbei wird zuallererst überprüft, welche Artikel schon einmal verarbeitet wurden. Diese werden verworfen. Die übriggebliebenen, also neuen Artikel werden in eine allgemeinere Repräsentation für Neuigkeiten und ihre Metadaten gebracht, die ich mir überlegt habe. Diese ist auch wieder eine Baumstruktur und sieht aus wie in Abb. 4.

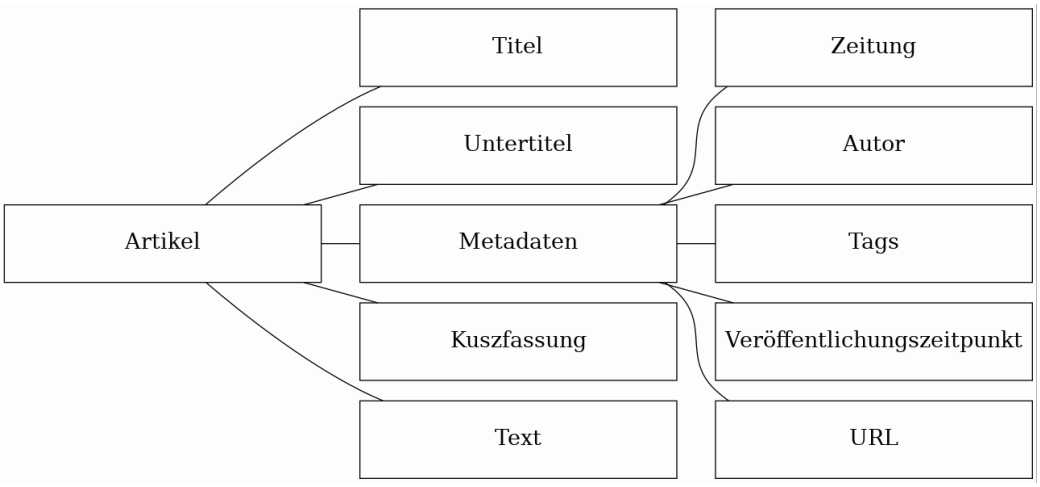

Abb. 4: Die Datenstruktur der zwischenrepräsentation der Artikel

Diese Umwandlung ist nötig, da der RSS-Standard zwar die grobe Struk-

tur und ihre Repräsentation als XML-Datei spezifiziert, letzten Endes jeder Nachichtenanbieter allerdings doch nicht ganz kompatible Feeds ausliefert. Diese kleinen Unterschiede werden hier also angeglichen, damit die weitere Verarbeitung leichter vonstatten gehen kann, und in der weiteren Verarbeitung keine Unterschiede mehr beachtet werden müssen.

Ein weiteres Problem der RSS-Feeds ist, dass sie nur Kurzfassungen der Artikel enthalten. In der Umwandlung müssen also noch die vollständigen Artikel von der Internetseite des Nachrichtenanbieters heruntergeladen und der Volltext aus dieser extrahiert werden. Hierzu wird zuerst das HTML-Dokument der Seite "geparsed". Aus dem "geparsten" HTML-Dokument werden nun die relevanten Teile, zu denen z.B. nicht die Kopf- oder Fußzeile gehören, mithilfe sogenannter "CSS-Selektoren" identifiziert. Hiernach werden alle HTML-Elemente dieser Bereiche in reinen Text umgewandelt.

Architektonisch ist der Datensammler ein Programm, in das weitere kleinere Module "hereingesteckt" werden, die die einzelnen Nachrichtenseiten ansprechen (Siehe Abb. 5).

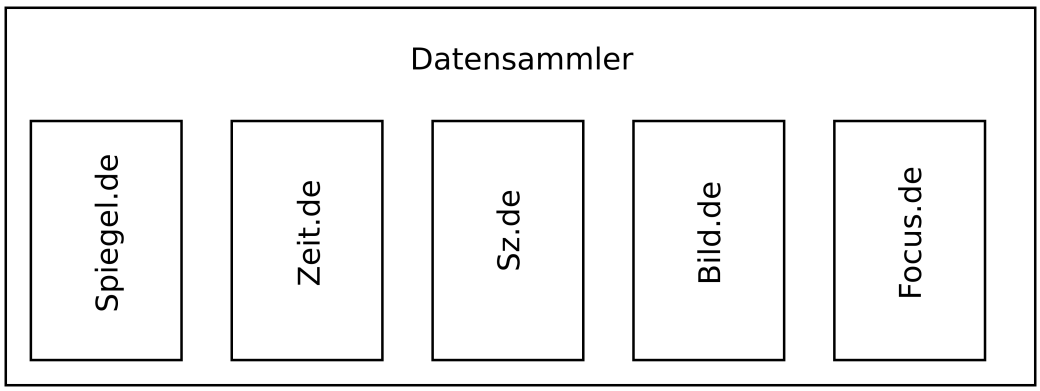

Abb. 5: Der Datensammler und seine Module

Am Ende der Datenaquirierung werden die gesammelten Daten an die nächste Stufe weitergegeben, in der die Daten gespeichert werden.

4.2 Speicherung

Die nun gesammelten Daten müssen vor der weiteren Verwendung ersteinmal strukturiert zwischenggespeichert werden. Dies geschieht in einer Datenbank.

Ich habe mich dafür entschieden, eine sogenannte “NoSQL-Datenbank” zu verwenden, da diese flexiblere Abfragemethoden ermöglichen. “NoSQL” steht hierbei dafür, dass die Datenbank nicht über die Datenbanksprache SQL angesprochen wird, wie es normalerweise der Fall ist, sondern eine andere Abfragemöglichkeit bietet. Relativ früh habe ich mich dafür entschieden, dass ich meine Datenbankabfragen als “MapReduce”-Funktionen formulieren möchte.

4.3 MapReduce

Das MapReduce-Verfahren hat einige große Stärken, wie z.B. die hohe Parallelisierbarkeit und die damit verbundene hohe Geschwindigkeit, bei gleichzeitig hoher Flexibilität. MapReduce ist ein von der Firma Google eingeführtes Programmiermodell, welches wie folgt funktioniert (Vgl. Abb. 6):

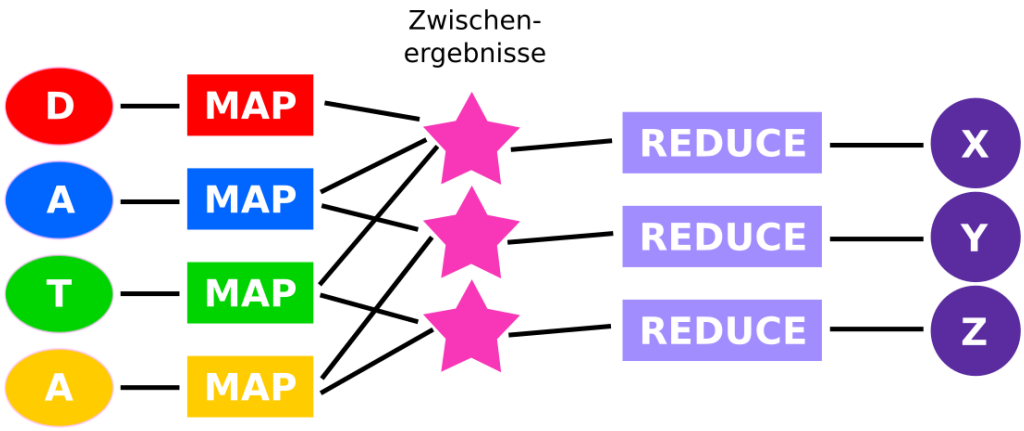

Abb. 6: Der MapReduce prozess als Grafik

1. Am Anfang des Prozesses steht eine Menge aus n Eingangsdatensätzen. In meinem Fall sind das die Zeitungsartikel, die als Objekte mit der oben beschriebenen Datenstruktur vorliegen. Für jeden dieser Artikel wird jetzt die Map-Funktion ausgeführt. Diese ordnet jedem Artikel n Schlüssel-Wert-Paare zu. In dem Fall, dass wir zu jedem vorkommenden Wort die Häufigkeit bestimmen wollen, ordnet die Map-Funktion also jedem Artikel die Menge der darin enthaltenen Wörter zu. Da diese Funktion auf jeden Artikel angewandt wird, kann

man sich in diesem Fall den gesammten Map-Prozess als eine Funktion vorstellen, die die Menge aller Artikel der Menge der darin enthaltenen Worte zuordnet. Der hierzu zugehörige Code dieser Map-Funktion ist:

```
function map() {
  var text = this.text.replace(/[^A-Za-zÄäÖöÜüß ]/g, " ");   // entferne alle überfüssigen Satzzeichen, wie .,?!-
  var words = text.split(" ");   // Teile den text in Wörter
  for (var i = 0; i < words.length; i++) {   // für jedes Wort
        var word = words[i];
        if(word) {
           emit(word, 1);   // füge der Ausgabemenge das Wort hinzu
        }
  }
}
```

2. Im nächsten Schritt werden Elemente mit gleichen Schlüsseln gruppiert. Nach diesem Schritt liegt also eine Menge aus Schlüssel-Wertmengenpaaren vor (Auch wenn ich immer wieder das Wort Menge verwende, meine ich eigentlich Multimengen, da es relevent ist, wie oft ein bestimmtes Element in der Menge vorkommt [@asdf]). Dieser Schritt ist, anders als die anderen beiden Schritte, bei allen anderen Abfragen gleich. Da in unserem Beispiel das Wort als Schlüssel verwendet wurde, sähe eine beispielhafte Menge nach diesem Schritt wie folgt aus, was stark an eine Strichliste erinnert:

```
[
   "der": {1; 1; 1; 1; 1; 1; 1; 1; 1},
   "die": {1; 1; 1; 1},
   ...
]
```

3. Im dritten und letzten Schritt wird für jeden Schlüssel die sogenannte Reduce-Funktion angewandt. Diese reduziert die Mengen, die den Schlüsseln zugeordnet sind, auf einen Wert. In unserem Beispiel fällt die Reduce-Funktion relativ einfach aus, da sie nur die Elemente der Menge aufsummieren muss:

```
reduce(key, values) {
```

```
2   return values.reduce((previousValue, currentValue) =>
        currentValue + previousValue);
3 }
```

4. Der vierte und letzte Schritt gehört eigentlich nicht mehr zum MapReduce-Verfahren. Dieser wird nach dem MapReduce-Verfahren ausgeführt und dient dazu, die Ergebnisse zu sortieren und eventuell Feinheiten zu verbessern. So ist es zum Beispiel möglich, selten genutzte Worte oder sogenannte Stoppworte in diesem Schritt auszusortieren. Stoppworte sind häufig auftetende Worte, die keine Relevanz für die Erfassung des Dokumenteninhaltes haben. Hierfür verwende ich verschiedene Stoppwortlisten, die von Sprachforschern erstellt worden sind, zusammen mit eigenen Ergänzungen. Der hierzu gehörige Code, der die Wortliste filtert, sieht wie folgt aus:

```
1 function filter(data) {
2   return data.filter(word =>
        stopwords.indexOf(word["_id"].toLowerCase()) < 0)
3 }
```

   Des weiteren wird in diesem Schritt versucht, Wörter mit dem gleichen Stamm, und somit mit der gleichen Bedeutung zusammenzuführen, auch wenn diese unterschiedliche Endungen haben. Ein Beispiel hierfür ist, dass die Wörter “Trump” und “Trumps” zusammengezählt werden. Hierbei wird immer das kürzeste Wort behalten, da dies meist die Grundform ist. Der Code hierfür ist nicht ganz so einfach:

```
1  function word_merge(list) {
2  modList = list.slice();
3  modList.forEach((nowItem, nowCount, nowObject) => {
4      var regex = new RegExp("^" + nowItem["_id"].toLowerCase() +
           '.{0,2}','g');
5      nowObject.forEach((item, index, object) => {
6          if(nowItem["_id"].toLowerCase() ==
               item["_id"].toLowerCase()) return;
7          checkWord = item["_id"].toLowerCase();
8          if(checkWord.match(regex)) {
9              nowObject[nowCount]["value"] += item["value"];
10             object.splice(index, 1);
```

```
11          }
12      });
13 });
14 return modList
15 }
```

4.4 Datenbank

Nachdem klar war, wie die Abfragen formuliert werden sollten, habe ich verschiedene Datenbanken in Betracht gezogen. Zuerst habe ich angefangen, mit der Datenbank "MongoDB" zu arbeiten, störte mich aber sehr stark an deren Komplexität. Außerdem bietet MongoDB keine Möglichkeit, diese über das HTTP-Protokoll anzusprechen, was für die Visualisierung allerdings sehr wichtig ist. Also sah ich mich nach anderen Alternativen um und fand "CouchDB", welche vom Apache-Projekt entwickelt wird. Diese Datenbank erfüllte die meisten meiner Anforderungen relativ gut, weshalb ich meinen gesamten, bis dahin existierenden Code an CouchDB anpasste. Nach einiger Zeit des Testens stellte sich allerdings heraus, das CouchDB wahrscheinlich zu langsam sein würde und ein unpassendes Rechteverwaltungssystem mitbringt, was die Arbeitsersparnis im Gegensatz zu MongoDB zunichte machen würde. In Folge dieser Erkentnis entschied ich mich dazu, meine Software zurück auf MongoDB zu portieren und für diese eine HTTP-Schnittstelle zu entwickeln.

4.5 Middleware

Diese HTTP-Schnittstelle ist gewissermaßen das Bindeglied zwischen der Datenbank und der Visualisierung, weshalb es im nachfolgenden "Middleware" genannt wird. Sie nimmt HTTP-Anfragen von der Visualisierung ("Frontend") entgegen, leitet diese an die Datenbank weiter und schickt die Antwort der Datenbank an das Frontend zurück. Insofern könnte man sie auch als Übersetzer zwischen verschiedenen Protokollen verstehen. Eine Übersetzung ist notwendig, da ich die Visualisierung im Webbrowser implementiert habe, und im Browser nur sehr wenige Schnittstellen verfügbar sind.

HTTP ist eine dieser wenigen Möglichkeiten und bietet sich an, da es verhältnismäßig einfach zu implementieren ist.

4.6 Visualisierung

Ich habe mich entschieden, die Visualisierung im Browser umzusetzen. Dies lag hauptsächlich daran, dass ich bereits einige der verwendeten Technologien kannte, und somit die Einstiegshürde niedriger war. Außerdem gibt es für den Browser und die damit verbundenen Technologien bereits sehr gute Bibliotheken zur Datenvisualisierung, auf die ich für meine Arbeit zurückgreifen konnte. Ein weiterer, nicht unwichtiger Punkt, ist die einfache Portabilität von Programmen, welche im Browser laufen: Sie können auf fast jedem Computer, unabhängig von Betriebssystem oder Prozessorarchitektur, innerhalb von Sekunden aufgerufen und ausgeführt werden.

Eines der Hauptziele bei der Konzeption und Entwicklung der gesamten Analysesoftware war die Flexibilität: Es sollte möglichst schnell und einfach möglich sein, verschiedene Analysen über den Datensatz durchzuführen. Dies bezog sich nicht nur auf vorgefertigte, bei der Entwicklung bedachte Analysen, sondern auch zukünftige Ansätze. So sollte es einfach möglich sein, eigene neue Analysen über den Datensatz durchzuführen. Dies führte zu der konzeptionellen Endscheidung, dass die Analyse durch den Quellcode des Visualisierungsmoduls vorgegeben sein sollte und alle anderen Zwischenschritte, wie die Middleware oder die Datenbank nur auf die Anfragen des Analysemodul hören sollten. Dies hat zur Folge, das ein großer Teil der Komplexität des Codes im Frontend liegt.

Jede Visualisierung fragt am Anfang die benötigten Daten bei der Middleware über MapReduce-Anfragen an. Die einzelnen Funktionen, die hierzu benötigt werden, werden also in dem Code der Visualisierung definiert und hängen davon ab, welche Daten aus dem Datensatz benötigt werden. Dieser Vorgang kann mehrere Male wiederholt werden, um verschiedene Informationen über verschiedene Aspekte des Datensatzes zu bekommen. Hiernach werden die nun vorliegenden Daten weiterverarbeitet. Sie können also gefiltert, sortiert oder miteinander verrechnet werden. Liegen die Daten nun in der gewünschten Form vor, kann die eigentliche Visualisierung beginnen.

Um die nun aufbereiteten Daten zu visualisieren, gibt es viele verschiedene

Möglichkeiten. Ich habe mich dazu entschieden, die Softwarebibliothek "d3.js" zu verwenden, welche es ermöglicht, Verbindungen zwischen Datenmengen und Elementen im Browser herzustellen. Das Aussehen dieser Elemente kann nun mithilfe von gängigen Techniken, wie "CSS" angepasst werden. Für komplexere Visualisierungen werden auch andere Bibliotheken verwendet, um z.B. die einzelnen Positionen der Worte zu berechnen. Mehr zu den einzelnen Visualisierungen ist im Abschnitt "Die verschiedenen Visualisierungen" zu finden.

read; konkreter; bild

4.7 Zusammenführung

Da die gesamte Software modular aufgebaut ist, besteht sie aus vielen kleinen

read

Komponenten, die spezialisiert und alleine nicht sehr nützlich sind. Diese kleinen Komponenten nennt man in der Softwareentwicklung "Microservices". Um aus diesen kleinen Microservices eine funktionierende Software zu erhalten, muss man diese jeweils in einer passenden Umgebung starten und richtig miteinander verbinden. Um dies zu bewerkstelligen, habe ich das Software-Paket "Docker" verwendet. Dieses erlaubt es zu allererst, die einzelnen Umgebungen für jeden Microservice zu beschreiben. Diese enthalten zum Beispiel die verschiedenen Programmbibliotheken, die ich in den einzelnen Teilen verwandt habe, oder einen Webserver, um den Visualisierungscode an den Webbrowser auszuliefern. Eine solche Umgebung mit allen darin enthaltenen Programmen nennt man "Container". Diese Container erstellt man immer auf der Basis eines anderen "Basiscontainers". Dies sorgt dafür, das man sich nicht immer um alles innerhalb eines Containers kümmern muss, also zum Beispiel nicht immer selber das Betriebssystem installieren muss, sondern nur noch die eigenen Komponenten hinzufügen muss. Dies geschieht automatisiert über eine Datei, die jedem Microservice beigefügt ist, dem sogenannten "Dockerfile". Auch ist es möglich komplett, vorgefertigte Container zu verwenden, was ich z.B. für die Datenbank nutze.

Die einzelnen Container, die nun die komplette Umgebung enthalten, die die jeweiligen Microservices zum laufen benötigen, müssen nun nur noch alle richtig verbunden werden. Dies geschieht mit einem Programm namens "Docker-compose". Diesem gibt man eine Datei, die den Zustand des gesamten gewünschten Systems beschreibt. Automatisiert wird aus diesem und allen Dockerfiles der Microservices dann genau das beschriebene System

erstellt und gestartet. Die Datei, die dieses System abbildet, sieht wie folgt aus:

```
1  version: '2'
2
3  services:
4     data-collectors:
5        build: data-collectors/
6        links:
7           - mongodb
8     backend:
9        build: backend/
10       links:
11          - mongodb
12
13    frontend:
14       build: frontend/
15       links:
16          - backend
17       ports:
18          - "80:80"
19
20    mongodb:
21       image: "mongo:latest"
```

Auch hier sieht man wieder sehr schön die Struktur und die Modularität der Software und ihre einzelnen Dienste.

Dieses System, das alle ständig laufenden Komponenten enthält, wird nun auf einem Server gestartet und ermöglicht es, sowohl den Code der Visualisierung als auch Auskünfte über den Datensatz zu bekommen.

5 Die verschiedenen Visualisierungen

Letzten Endes habe ich mich entschlossen, vier verschiedene Analysen zu implementieren. Diese geben verschiedene Informationen über den Datensatz aus und sind alle sehr unterschiedlich komplex. All diese sind jedoch nicht

trivial, da immer versucht werden muss, aus einem Fließtext, also einem Format, mit dem Computer eigentlich nichts anfangen können Informationen zu gewinnen und zu veranschaulichen. Hierbei ist eine besondere Herausforderung, dass dies alles passieren muss, obwohl der Computer eigentlich über kein versändniss von "Sinn" verfügt. Aus diesem Grund müssen statistische Verfahren als Hilfsmittel zur Hand gezogen werden, die es dem Menschen, der die Daten letzten Endes interpretiert ermöglichen Informationen aus der Datenmenge zu ziehen.

5.1 Häufigkeitstabelle

Die erste und einfachste Form der Visualisierung, die ich implementiert habe, ist die Häufigkeitstabelle (siehe Abb. 7). Sie gibt an, wie häufig ein bestimmtes Wort verwandt wurde. Sie ist eigentlich noch keine Visualisierung im engeren sinne, da sie nur aus einer Tabelle besteht, und die Daten nicht ansprechend aufbereitet. Sie ist dennoch sehr praktisch um sich schnell einen groben überblick über den Datensatz zu verschaffen. Dies habe ich auch sehr exzessiv bei der Entwicklung von Algorithmen wie z.B. dem zum zusammenführen ähnlicher Worte genutzt. Der Code, der der Wortzählung zu Grunde liegt ist bereits im Abschnitt "MapReduce" gezeigt und erläutert worden. Auch ist es sehr wichtig, dass Stoppworte herausgefiltert werden, ein relevantes Ergebniss zu produzieren.

5.2 Wortwolke

Die Grundidee meiner Halbjahresarbeit kam mir, als ich im Spiegel eine Wortwolke entdeckte. Diese Visualisierung, war also auch sehr naheliegend und konnte auf den gleichen Datensatz, wie die Häufigkeitstabelle zurückgreifen. Dieser wird nun nicht mehr als Tabelle ausgegeben, sondern als Wortwolke, in der öfter vorkommende Wörter Größer sind.

better images; search for specific events

word	count
Jahr	257
La	197
Uhr	151
EU	122
Polizei	110

Abb. 7: Die Darstellung der Worthäufigkeiten als Tabelle

Abb. 8: Eine Wortwolke vom 9. Januar 2017

5.3 Landkarte

Meine nächste Visualisierung sollte nun nicht nur die häufigkeit verwandter Wörter, sondern auch ihren Zusammenhang darstellen. Ich wollte also, dass Wörter, die öfter zusammen vorkommen näher beieinander stehen. Dies habe ich erreicht, indem ich zuerst, wie bei der Häufigkeitstabelle und der Wortwokle die Häufigsten wörter abgefragt habe. Danach habe ich eine zweite Abfrage geschrieben, die die Distanz der Häufigsten n Wörter zueinander in den Artikeln feststellt.

```
1  function map() {
2      var givenWords = __marker__;
3      var distance_function = __marker2__;
4      var words = this.text.replace(/[^A-Za-zÄäÖöÜüß ]/g, "
           ").split(" ").filter(w => w);
5
6      // create dict with {"word": ['o', 'c', 'c', 'u', 'r', 'e',
           'n', 'c', 'e']}
7      var wordOccurrences = {};
8      givenWords.forEach(word => {
9          var occurrences = [];
10         for(var i = 0; i < words.length; i++) {
11             if(word == words[i]) {
12                 occurrences.push(i);
13             }
14         }
15         wordOccurrences[word] = occurrences;
16     });
17
18     givenWords.forEach(w => {
19         givenWords.forEach(v => { // every word combination is
               covered as w and v
20             if(w != v && wordOccurrences[w] &&
                   wordOccurrences[v]) {
21                 list = [];
22                 wordOccurrences[w].forEach(n => {
23                     wordOccurrences[v].forEach(m => {
24                         list.push(distance_function(n, m));
```

```
25                      });
26                  });
27
28                  strength = list.reduce((a, b) => a + b, 0);
29                  if(strength != 0) {
30                      arr = [w, v].sort();
31                      emit(arr[0] + "." + arr[1],
                            JSON.stringify([strength, 1]));
32                  }
33              }
34          });
35      });
36  };
```

5.4 Zeitreihe

Natürlich kann man sich in dem gesammelten Datensatz auch ganz andere Parameter angucken und muss sich nicht immer nur den Zustand zu einem Zeitpunkt betrachten. So ist es zum Beispiel möglich Aussagen über die veränderung verschiedener Parameter über die Zeit treffen. Genau dies habe ich nun also in meiner nächsten Visualisierung gemacht.

Zuerst habe ich nur die Anzahl der pro stunde veröffentlichten wörter über die Zeit betrachtet (siehe Abb. 10).

5.5 Vergleich verschiedener Zeitungen

Als letztes habe ich noch eine Relativ einfache Analysemöglichkeit gebaut, die es ermöglicht,

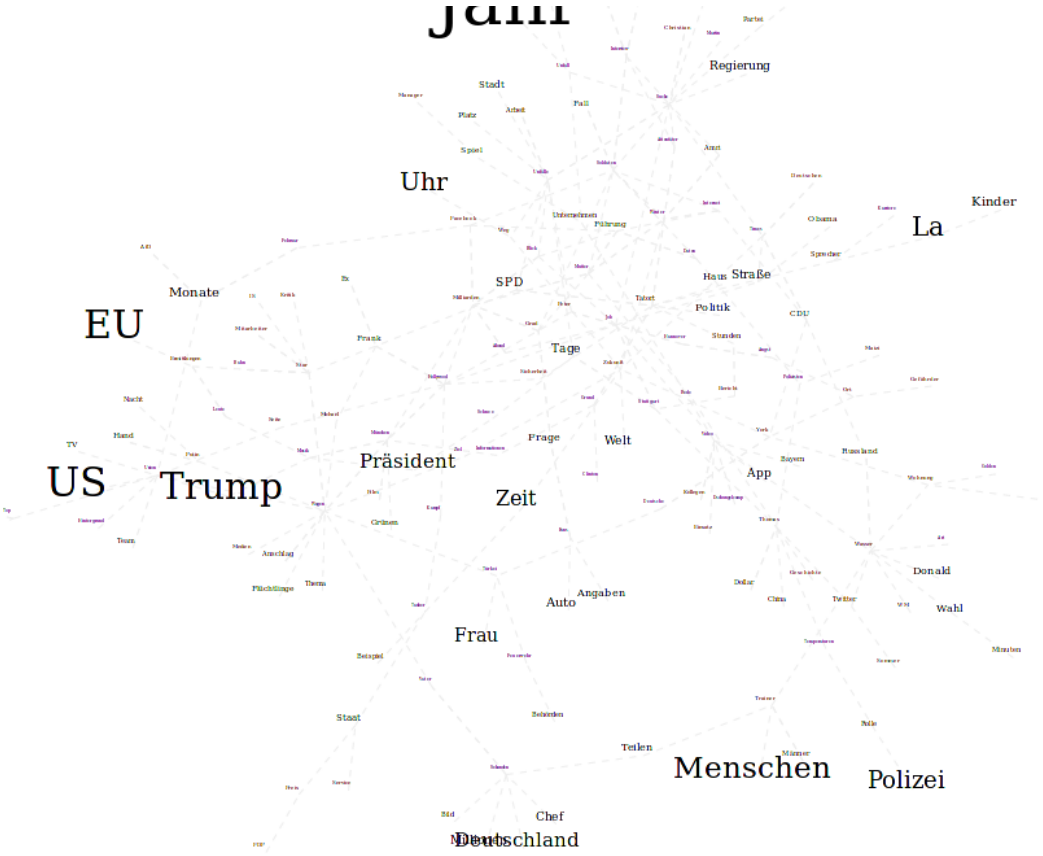

Abb. 9: Die Nachrichtenlandkarte vom 9. Januar

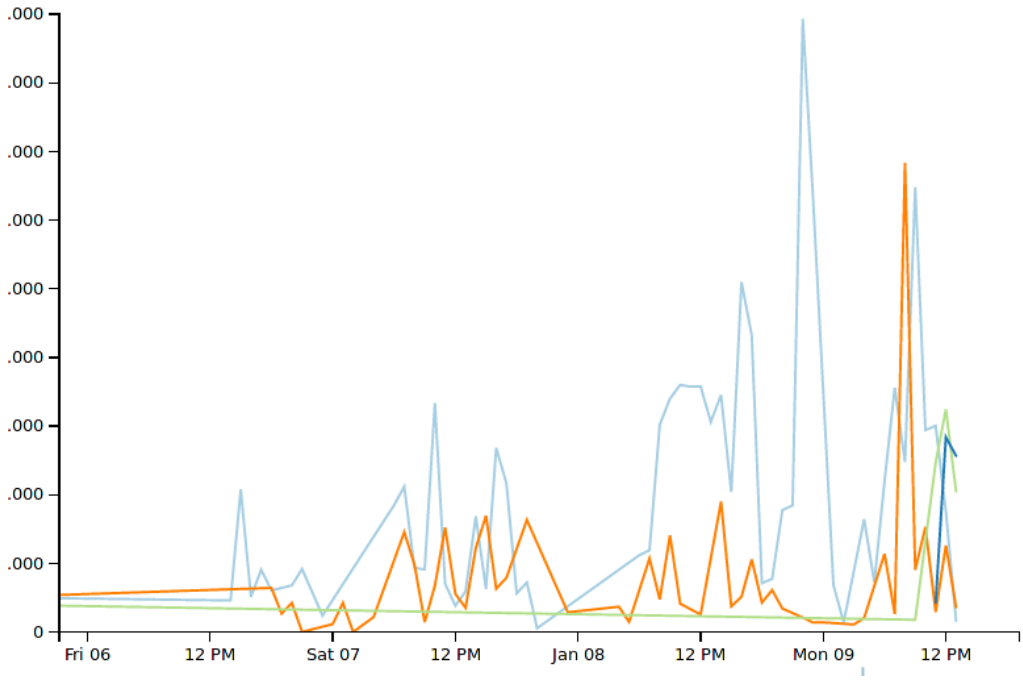

Abb. 10: Die Anzahl der geschriebenen Wörter pro Stunde über die Zeit

6 Fazit

7 Quellen