HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL

FHO Fachhochschule Ostschweiz

Studienarbeit, Abteilung Informatik

myArticle.io

Hochschule für Technik Rapperswil

Frühlingssemester 2015
29. Mai 2015

Autoren:	Kirusanth Poopalasingam & Martin Stypinski
Betreuer:	Prof. Dr. Josef M. Joller
Arbeitsperiode:	16.02.2015 - 29.05.2015
Arbeitsumfang:	240 Stunden, 8 ECTS pro Student
Link:	`http://www.myArticle.io/`

Eigenständigkeitserklärung

Wir erklären hiermit,

- dass wir die vorliegende Arbeit selber und ohne fremde Hilfe durchgeführt haben, ausser derjenigen, welche explizit in der Aufgabenstellung erwähnt sind oder mit dem Betreuer schriftlich vereinbart wurden,
- dass wir sämtliche verwendeten Quellen erwähnt und gemäss gängigen wissenschaftlichen Zitierregeln korrekt angegeben haben,
- dass wir keine durch Copyright geschätzten Materialien (z. B. Bilder) in dieser Arbeit in unerlaubter Weise genutzt haben.

Rapperswil, den 29. Mai 2015

Kirusanth Poopalasingam

Martin Stypinski

Danksagung

Zunächst möchten wir uns an dieser Stelle bei all denjenigen bedanken, die uns während der Anfertigung dieser Arbeit unterstützt und motiviert haben.

- Prof. Dr. Josef. M. Joller für die Unterstützung während der gesammten Arbeitszeit und den Freiraum der uns gewährt wurde.
- Prof. Dr. Andreas Müller für die wertvollen Ideen und die Erklärungen im Bereich des Bayes-Klassifikators.
- Sainuka Poopalasingam für die Korrektur der Rechtschreibfehler.
- Jessica Stypinski für das Überarbeiten der Arbeit.
- Und all denjenigen die uns seelisch und psychisch in dieser Zeit zur Seite standen.

Abstract

Wie gelangen wir in der heutigen Zeit am schnellsten an Informationen?
Das Internet ist immer dichter gepackt mit Informationen. Suchmaschinene wie Google, Bing und Yahoo versuchen zwar einen einfacheren Zugang zu dieser Informationsflut zu gewährleisten, jedoch ist die Suche und das Finden von aktuellen und vergangenen Ereignissen äusserst schwierig.
Ziel dieser Arbeit ist es, einen Einstieg in die funktionale Programmierung mit Haskell zu finden. Da das Erlernen einer neuen Programmiersprache einer eher trockenen Angelenheit entspricht, wurde bewusst ein Thema gewählt, welches uns genug herausfordern und uns neue viele Erfahrungen bringen sollte.
Das Resultat der Arbeit ist eine Internetsuchmaschine, die für spezifische zeitgebundene Ereignisse verschiedene Zeitungsquellen und Enzyklopädien miteinander verknüpft. Der Leser soll seine zu recherchierenden Informationen oder Themen auf einen Blick erhalten können und diese unter verschiedenen Betrachtungswinkeln nachlesen können.
Die Erkenntnisse und das Produkt der Arbeit soll die Realisierbarkeit einer solchen Arbeit aufzeigen.

Management Summary

Ausgangslage

Gute Informationsquellen und Zeitungshäuser sind in grossen Mengen im Internet repräsentiert. Gute Artikel sind sehr detailiert, jedoch fehlen dem Leser aber oft die Zusammenhänge. Diese Tatsache benutzten wir als Ausgangslage, um ein Projekt zu realisieren, welches dem Leser die Möglichkeit gibt die gesuchte Information in einem groben Zusammenhang vorzufinden. Eigentlich war dieser Gedanke nur der Vorwand für die Arbeit. In Wirklichkeit war die Idee sich als Team in eine funktionale Programmiersprache einzudenken und diese von Grund auf zu erlernen. Es wurde auf Haskell gesetzt, da es interessante Konzepte und Abstraktionen verwendet, welche in imperativen Sprachen nicht vorhanden sind. Ausserdem zwingt Haskell rein funktional zu programmieren, ohne in alte objektorientierte Muster zurückzufallen.

Ergebnisse

Das Hauptziel der Arbeit war es, Haskell zu erlernen und das Gelernte dabei in ein Produkt umzusetzten. Schnell war klar, dass ein finales Produkt,so wie es gewünscht war, doch etwas hochgegriffen war. Stattdessen kann man feststellen, dass es ein interessantes Proof of Concept geworden ist. Insgesamt muss man sagen, dass letzendlich einfach zu viele Hürden in den neuen Technologien gefunden wurden, die eine schnelle Realisierung schwierig machten.

Dem Benutzer wird bei `http://www.myarticle.io/` eine Webapplikation gezeigt, die eine Vision darstellt, wie in Zukunft Zeitungen gelesen werden könnten. Es soll aufzeigen, wie einfache Fakten im Kontext zwischen Enzyklopädien stehen und in welchen Medien bereits ähnliche Artikel erschienen sind. Eine einfache Suchmaske im Stil von Google soll dem User eine möglichst einfache Handhabung anbieten.

Ausblick

Die Möglichkeiten einer potenziellen Weiterentwicklung sind enorm gross. Einerseits sollte ganz klar ein Schwerpunkt auf dem User Interface und der Bedienung gelegt werden. Weitere Funktionen, die eine hohe Attraktivität für den Benutzer haben, können einfach implementiert werden. Aus einem technologischen Blickwinkel, wären zum Beispiel ein noch besserer Algorithmus und detailiertere Ansätze einer Clusteranalyse oder eines Bayes-Klassifikators möglich. So könnten wir uns vorstellen, dass eine 'Vorgruppierung' der Artikel interessant wäre um unter anderem die Performance massiv zu verbessern. Insgesamt müssen wir selbstkritisch feststellen, dass es vermutlich eine andere Technologie gibt, die für dieses Produkt besser geeignet wäre. Die Arbeit stand ganz klar im Zusammenhang mit Haskell und war ausserordentlich lehrreich, jedoch sollte aber für eine Weiterentwicklung der Technologie-Stack noch einmal überdacht werden.

Aufgabenstellung

Betreuer

- Prof. Dr. Josef. M. Joller, Dozent für Informatik HSR

Ausgangslage, Problembeschreibung

Das übergeordnete Thema dieser Arbeit ist das Suchen von Informationen zu einem Thema, also nicht nur das Suchen anhand eines Schlüsselwortes, sondern zudem Hinweise auf Dokumente und Informationen, welche mit dem Schlüsselwort semantisch zusammenhängen. Klassische Suchmaschinen wie Google, Yahoo oder Bing liefern in der Regel Index-basierte Listen mit Links auf Web-Dokumente, in denen das Schlüsselwort vorkommt. Ein einfacher Test mit Google zeigt, dass in der Regel viel zu viele Links aufgelistet werden und die Relevanz der Ergebnisse nicht immer klar ersichtlich ist, obschon gemäss Google die Links nach Relevanz absteigend aufgelistet werden. Verschiedene andere Suchmaschinen-Ansätze wurden bereits veröffentlicht, oft in einem experimentellen Stadium. Das Thema ist kaum analytisch fassbar. Mehrere Ansätze basieren auf Ontologien (sematischer Web). Aber keiner der Ansätze hat sich als dominant herausgestellt.

Die bisherigen Ansätze wurden in der Regel mithilfe klassischer Programmieransätze, sprich Perl bis Python, implementiert. Da heute zunehmend funktionale Ansätze in der Programmierung verfolgt werden, bietet es sich an, das Thema mithilfe einer funktionalen Programmiersprache anzugehen, beispielsweise mit Haskell.

Aufgabenstellung

Grobziel der Arbeit ist es mit einem überschaubaren Projekt einen Einstieg in die funktionale Programmierung auf der Basis von Haskell zu erlauben. Das Projekt muss eine bestimmte Komplexität besitzen, also über die einfachsten Listen- und Basis-Funktionen von Haskell hinausgehen und die Komponenten:

1. Benutzeroberfläche: damit bewegt man sich automatisch auf der Ebene der reaktiven Programmierung;
2. Businesslogik: verarbeiten (selektieren, verdichten, umformen) von Informationen;
3. Datenhaltung: Datenhaltung (einfügen, selektieren, aktualisieren) von Informationen in einer Datenbank;
4. Kommunikation: entfernte Informationen lesen und verarbeiten.

Benutzeroberfläche

Die Benutzereingabe soll möglichst einfach erfolgen können. Nach Eingabe eines Suchbegriffs werden „benachbarte“ Informationen - Web-Meldungen - zum Thema gesucht und gewichtet, je nach Relevanz zum Suchbegriff. Dazu muss ein Relevanzmass definiert werden.

Die Programmierung der Benutzeroberfläche kann im Falle eines Zeitproblems auch in Java (typischerweise JavaFX) oder JavaScript erfolgen. Benutzeroberflächen lassen sich in funktionalen Programmiersprachen, also hier in Haskell, nur mithilfe reaktiver Frameworks realisieren. Diese sind recht umfangreich (Netwire, Reactive Banana) und recht komplex, da die Zustände in Monaden und Arrows gekapselt werden müssen.

Auf der andern Seite bietet sich hier die Möglichkeit sich mit diesem Thema auseinanderzusetzen.

Businesslogik

Dies umfasst die Bewertung der im Web gefundenen Informationen / Dokumente. Basis dafür ist das Relevanzmass: wie „dicht“ ist eine Meldung bei einem gesuchten Suchbegriff?

Die Programmierung der Businesslogik ist eher klassisch und unterscheidet sich nicht sehr von deren Umsetzung in OO oder imperativen Programmiersprachen.

Datenhaltung

Die Suchergebnisse müssen in einer Datenbank gespeichert und anschlissend mit der Businesslogik ausgewertet werden. Datenbankzugriffe aus Haskell sind für verschiedene DBs vorhanden. Dieser Layer sollte daher wenig Probleme machen.

Kommunikation

Da die relevanten Dokumente aus dem Web stammen, muss ein einfacher Crawler gebaut werden. In Haskell sind verschiedene Crawler realisiert worden, inklusive Parallelisierung. Hier könnte eine Challenge sein, Haskell Concurrency kennen zu lernen.

Zur Duchführung

Die Bearbeitung der Aufgabe setzt eine Einarbeit in die funktionale Programmierung, Haskell, voraus undzwar auf einem vertiefenden Level (Monaden, allenfalls Arrows).

In der Regel findet ein wöchentliches Meeting statt. Vorgängig zum Meeting sollten offene Punkte schriftlich per Mail bekannt gemacht werden. Die Ergebnisse, sofern es sich um Entscheidungen handelt, werden in Protokoll-Dokumenten festgehalten.

Dokumentation und Abgabe

Die Dokumentation muss sich an den Richtlinien der HSR orientieren, mit folgenden Abweichungen:

1. der persönliche Bericht muss in einem separaten Dokument abgegeben werden (auf der CD).
2. der Projektplan kann recht grob sein und muss nur wesentliche Meilensteine umfassen, da die Arbeit einen innovativen Charakter hat und somit wenig planbar ist.
3. die Dokumentation ist wie üblich in 3 Exemplaren (als CD) abzugeben.

Termine

Die Termine ergeben sich aus den Semesterterminen der HSR (Bachelorstudium).

Inhaltsverzeichnis

1 Technischer Bericht

1.1 Einführung

Die Informationsdichte im Internet nimmt stetig zu. Dank Suchmaschinen wie Google, Bing oder Yahoo ist es zwar möglich an Informationen zu gelangen, jedoch ist es oft schwierig in kurzer Zeit sich über grössere zeitgebundene Kontexte zu informieren.
Zeitungen und Verlagshäuser haben nach wie vor eine tragende Bedeutung was die Berichterstattung betrifft. Diese zeigen grundsätzlich punktuelle Ereignisse aus einem bestimmten Betrachtungswinkel, oft fehlt es jedoch dem Leser an Verständnis für die gesamten Spannweite des Ereignisses.
Journalisten verbringen viel Zeit mit der Rechereche und Aufarbeitung diverser Quellen, um uns aktuelle, spannende und informative Artikel zu präsentieren. Jedoch sind diese oft auf das wesentliche komprimiert und ohne Hintergrundinformationen sind diese nur schwer einzuordnen.

1.1.1 Vision

Die Vision entstand während einer Kaffeepause. Wir haben über die Entwicklung einer politischen Situation diskutiert, die in einem Zeitungsartikel erwähnt wurde. Leider war es uns nicht möglich Hintergrundinformationen zu diesem Ereignis zu finden, und nur mithilfe von Wikipedia konnte dann der Hergang rekonstruiert werden. Trotz der Informationsfülle im Internet war es uns schlicht und einfach nicht möglich die richtigen Informationen schnell und kompakt in einem Kontext zu finden.
Somit war die Idee für diese Arbeit geboren: Genau diese Situation brachte uns auf die Idee einen 'kontextbezogenen Artikel'-Service anzubieten. Es sollte möglich sein, mithilfe des Internets und einigen ausgewählten Quellen, Artikel zu finden, diese zu bewerten und in einen Kontext zu setzen. Wie einfach wäre es, etwas zu recherchieren oder morgens die Zeitung zu lesen, wenn bereits die nötigen Hintergrundinformationen zu den wichtigen Geschehnissen mitgeliefert werden?

1.1.2 Ziele

Das Hauptziel dieser Arbeit ist primär nicht die Entwicklung eines Produktes oder die Umsetzung der Vision. Diese kam erst viel später dazu. Wir haben unseren Betreuer Prof. Dr. Josef. M. Joller in erster Linie kontaktiert, um eine Studienarbeit im Bereich der funktionalen Programmierung zu machen.

1.1.3 Rahmenbedingungen

Wir haben uns aus Gründen der Realisierbarkeit oder der Interessen auf folgende Rahmenbedingungen geeinigt:

- **Funktionale Programmierung:**
  Das Ziel vor Augen war ganz klar die gesamte Arbeit in Haskell zu schreiben, jedoch hat uns unser Betreuer Prof. Dr. Josef. M. Joller darauf hingewiesen, dass Java als Backupplan für spezifische Probleme eine erwägenswerte Alternative sei.

- **Informationsquellen:**
  Da wir sehr früh verstanden haben, dass wir vermutlich keinen generischen Crawler für das gesamte Internet schreiben können, haben wir uns entschlossen **3 unterschiedliche Zeitungsquellen** in **englischer Sprache** zu wählen. Zusätzlich haben wir beschlossen Wikipedia hinzuzuziehen, da diese Quelle offen und einfach zu verwenden ist.

- **Technologien:**
  Da bereits Haskell und die Aufgabenstellung eine Herausforderung für sich darstellen, verzichteten wir auf sämtliche technologischen Experimente. Wir verwenden nur weitere Technologien, die uns weitgehend bekannt sind. Beispielsweise PostgreSQL als RDBMS.

1.1.4 Vorgehen und Strategie

Während der ganzen Arbeit wurde ein Credo zum Leitsatz: „Risiken frühzeitig erkennen und verstehen“. Durch zahlreiche Projekte aus privatem und geschäftlichem Umfeld, ist klar geworden, dass Risiken nicht minimiert werden können, ohne diese genau zu kennen und zu verstehen. Diese Strategie hatte sich bereits während des Software Engeering Projektes als das wegweisende Lösungskonzept bewiesen. Daher wurde die Arbeit folgendermassen gegliedert:

1	Einarbeitung	Durch den verzögerten Projektstart und die turbulente administrative Phase, wurde die Zeit eingesetzt um ersten Kontakt mit Haskell zu knüpfen.
2	Strukturierung	Das Projekt umfasst viele Hürden und um diese sicher zu umschiffen, muss erst eine vernünftige Struktur geschaffen werden.
3	Proof of Concept	Anhand der Struktur können sehr schnell die Risiken herausgelesen werden. Das PoC soll helfen diese Riskien so früh wie möglich zu erkennen und Lösungsstrategien für die Umsetzung auszuarbeiten.
4	Produktifizierung	Da viele Ideen vorhanden sind und die Zeit relativ begrenzt, soll bewusst Zeit genommen werden um aus der Vision ein Produkt zu schaffen.
5	Implementation	Die Webapplikation soll implementiert werden und zur Produktreife heranwachsen.
6	Finalisierung	Diverse Abschlussarbeiten an der Dokumentation sollen noch vollzogen werden, Resultate aus der Implementation sollen festgehalten werden.

1.2 Umsetzung

Aus technischer Sicht ist die Umsetzung der Arbeit sehr spannend. Da das Ziel war ein Produkt in einer komplett neuen Programmiersprache zu entwickeln, sind einige Herausforderungen zu bewältigen gewesen. Die Umsetzung der gesamten Arbeit wurde zumeist in Haskell vollzogen. Für einige Details wurde SQL als sehr mächtige Abfragesprache ausgenutzt, um gewisse Teile performanter zu gestalten. Die Datenmenge stellte sich als eine zentrale Herausforderung während der gesammten Arbeit.
Als Kernthemen wurden folgende Problemstellungen bearbeitet:

- **Full Stack:**
  Es wurde ein komplett Webstack in Haskell implementiert. Angefangen bei der Datenbankanbindung bis hin zum Ausliefern der HTML Webseite durch einen Standalone Webserver
- **Datei Verarbeitung:**
  Im Verlauf der Arbeit wurde ein Tool geschrieben um Wikipedia Backup Dateien in unsere Datenbank einzulesen. Die Datenmenge von über 50GB erschwerte die Aufgabe deutlich.
- **Performance:**
  Es wurden verschiedene Methoden untersucht und experimentiert um die Performance von gewissen Prozessen in Haskell zu beschleunigen.

Die Umsetzung war sehr lehrreich, weil viele bekannten Konzepte und Erfahrungen in Haskell nicht gültig sind. Den grössten Vorteil, der in der Umsetzung ausgenutzt werden konnte, war die Tatsache, dass viel Projekterfahrung vorhanden war und die Risiken sehr gut abgeschätzt wurden. Am Ende kann auf eine lehrreiche Phase zurückgeblickt werden.

1.3 Resultate

Vom technischen Standpunkt ist das Resultat der Arbeit eine lehrreiche Zeit in einem komplett neuen Programmierparadigma. Haskell als Programmiersprache aus akademischem Umfeld verfügt über viele Konzepte, die während dieser Zeit angeschaut wurden. Es kann festgehalten werden, dass ein solider Einstieg gelungen ist, der Abstieg bis in die Tiefen der Sprache war jedoch aus Zeitgründen nicht möglich. Als 'Produkt' der Arbeit sind folgende Punkte eine detailierte Erwähnung wert, diese waren in der einen oder anderen Form ein Ziel der Arbeit:

- **Haskell als primäre Programmiersprache**
  Trotz diverser Schwierigkeit und vielen Umwegen, wurde versucht das Ziel, Haskell als primäre Programmiersprache einzusetzten, durchzuziehen. Es kann gesagt werden, dass Haskell die einzige vollwärtige Programmiersprache in dem gesamten Projekt ist. Für sämtliche produktiven Codes wurde Haskell verwendet. An manchen Stellen wurden komplexe SQL abfragen verwendet, um die Performance der Anwendung zu erhöhen und gewisse Berechnungen so nah wie möglich an der Datenhaltung zu vollziehen.

- **Text Klassifiezierung mittels naive Bayes**
  Mittels dem 'naive Bayes' Verfahren wurde ein Algorithmus entwickelt, welcher Texte vergleichen kann und bewerten kann, ob zwei Texte ähnlich sind.

- **Nachrichten Import aus Verlagshaus**
  myArticle.io bezieht Informationen von dem US-amerikanischen Verlagshaus 'The Guardian'. Dies sind teils tagesaktuelle Artikel und bilden den Grundstein der Informationssuche. Diese werden über Webservices abgefragt und dann in einer Datenbank abgelegt, weiterverarbeitet und indexiert.

- **Wikipedia als Big Data**
  Für die Hintergrundinformationen wurden Wikipedia Daten verwendet. Um die Daten vernünftig auswerten zu können, wurde bewusst aus Wikipedia.com-Backupdaten die wichtigsten Informationen entnommen und in die eigene Datenbank eingespielt. Der Umgang mit den Dateien, die bis über 50GB an Grösse haben, stellte sich als eine grosse Herausforderung heraus. Prozesse, die im Test einige Minuten dauerten und Prototypen welche gut funktionierten, brachten im Testbetrieb die Server bis an die Resourcelimite. Diese Herausforderung eröffnete einen guten Punkt, um Optimierungen in Haskell anzugehen.

Abschliessend und analog zum 'Management Summary' sollte an dieser Stelle gesagt werden, dass das Projekt einige Fragen oft mit dem Satz 'because we can' beantwortet hat. Viele Sachen waren augenöffnend aber nicht intuitiv. Sollte die Arbeit fortgesetzt werden, muss an dieser Stelle ein weiteres Mal erwähnt werden, dass Haskell möglicherweise nicht die beste Technologie dafür ist. Sollte die ganze technologische Erkenntnis aus dieser Arbeit auf einen Satz gebracht werden, so kann man festhalten, dass Haskell enorm mächtig ist und absolute Exisitenzberechtigung hat, jedoch muss man sagen, dass für einen kompletten Webstack deutlich bessere Technologien auf dem Markt verfügbar sind.

2 Einarbeitung

Bevor die Arbeit aufgenommen werden konnte, mussten zuerst die Schwerpunkte der Probleme festgestellt werden. Da diese Arbeit eine Möglichkeit darstellte, Haskell zu lernen, war eine spezfische Einarbeitung wichtig. Daher wurde dies als eine der Punkte für die Einarbeitung gewählt. Als zweiter Punkt wurde entschieden, dass die Textanalyse eine nähere Betrachtung benötigt. Somit wurde in der Einarbeitung auf diese zwei Punkte näher eingegangen.

2.1 Haskell

2.1.1 Vorkenntnisse

Da beide aus unterschiedliche Erfahrungen und Hintergründe haben, waren die Vorkenntnisse am Anfang der Arbeit sehr verschieden.

- **Kirusanth Poopalasingam**
  Aus Interesse an den funktionalen Konzepten hat er das Buch 'Learn You a Haskell for Great Good!' [Lip11] gelesen, und einige kleine Übungen gelöst, jedoch noch keine praktische Applikationen geschrieben.

- **Martin Stypinski**
  Haskell war ihm zwar ein Begriff, aber nur aus der kurzen Einführung im Modul Compilerbau und Programmierkonzepte.

Somit kann festgehalten werden, dass Haskell zwar für Beide ein Begriff war, jedoch das Wissen zuwenig vertieft um grosse Projekte durchzuführen.

2.1.2 Einarbeitung

Um einen schnellen Einstieg in Haskell zu bekommen, wurden die Vorlesungsunterlagen von Brent Yorgey benutzt, welche vollständig online verfügbar sind. [Yor15].
Dies war sowohl die Empfehlung von Herrn Prof. Dr. Josef. M. Joller und auch von der Haskell Community [All15]. Für die Vertiefung in die Konzepte, wurden die Bücher 'Learn You a Haskell for Great Good!'[Lip11] und 'Programming in Haskell'[Hut07] verwendet.
Anfänglich wurde versucht zu den einzelnen Themen auch die Übungen zu lösen. Doch nach einigen Kapiteln wurde darauf verzichtet. Dies lag nicht nur an der zeitlichen Limitierung, sondern auch daran, dass die Aufgaben zu theoretisch sind und auch gemässe Herrn Joller meist zu komplex.
Da die erwähnten Bücher nur auf theoretische Beispiele fokussiert sind, wurde während der Implementierung auch das Buch von Bryan O'Sullivan 'Real World Haskell' [Hut07] verwendet,

welches die für uns nötigen Themen wie Datenbankverbindungen und Webapplikationen gut erklärt.

2.2 Semantische Analyse

2.2.1 Einführung

Als Leser von Texten ist es sehr einfach festzustellen, ob diese in einem Zusammenhang stehen oder nicht. Der Mensch hat die Fähigkeit Zusammenhänge sehr schnell zu verstehen und somit die Artikel zu kategorisieren und Themenbereichen zuzuordnen. Dem Computer fällt das deutlich schwerer. Für unsere Aufgabenstellung ist es aber von zentraler Bedeutung Texte zu vergleichen und zu analysieren. Es muss ein Algorithmus gefunden werden um Texte zu bewerten und um eine qualitative Aussage über die Ähnlichkeit machen zu können.

2.2.2 Vorüberlegung

Schon vor einer intensiven Recherche wurde versucht ein Lösungsmodel zu erarbeiten. Bereits am Anfang war klar, dass die ersten Versuche mit sehr einfachen Mitteln umsetzbar sein müssen. Der erste Gedanke fiel deutlich auf eine Analyse ohne nähere Auswertung des Kontexts. Der Text sollte so analysiert werden, als wäre er eine Sammlung von Wörtern. Die Grammatik, die Semantik und die Syntax sollen vollkommen vernachlässigt werden. Sehr schnell wurde klar, dass Themen, Gruppen, Zusammenhänge und Nachbarschaften in die Richtung einer Clusteranalyse führen würden. Der erste und einfachste Versuch wurde gestartet mit folgenden 3 Sätzen:

1. Der Mann ist im Garten.
2. Der Mann geht mit dem Hund spazieren.
3. Der Hund ist im Garten

Diese 3 Sätze sind unabhängig voneinander und ohne Kontext. Sie sind grammatikalisch und semantisch in sich korrekt. Jedoch teilen sie einige gemeinsame Wörter. Diese drei Sätze bilden das erste Model. Angenommen es existiert eine Basis B für einen Raum R^3 dieser Vektorräume wäre diese:

$$B = \left\{ \begin{pmatrix} Mann & 0 & 0 \end{pmatrix}^T, \begin{pmatrix} 0 & Garten & 0 \end{pmatrix}^T, \begin{pmatrix} 0 & 0 & Hund \end{pmatrix}^T \right\}$$

Somit lassen sich die 3 Sätze als Vektoren in diesem Raum darstellen:

1. Der Mann ist im Garten: $\begin{pmatrix} 1 \\ 0 \\ 1 \end{pmatrix}$
2. Der Mann geht mit dem Hund spazieren: $\begin{pmatrix} 1 \\ 1 \\ 0 \end{pmatrix}$

3. Der Hund ist im Garten: $\begin{pmatrix} 0 \\ 1 \\ 1 \end{pmatrix}$

Naheliegend folgt nun die Überlegung, wie weit die Vektoren auseinander liegen und wie stark verwandt die Sätze sind. Da es sich um einen n-Dimensionalen Vektorraum handelt, ist das Skalarprodukt [KN02] (Cosinus-Ähnlichkeit) der wohl einfachste Weg um die Ähnlichkeit zweier Vektoren nachzuweisen.

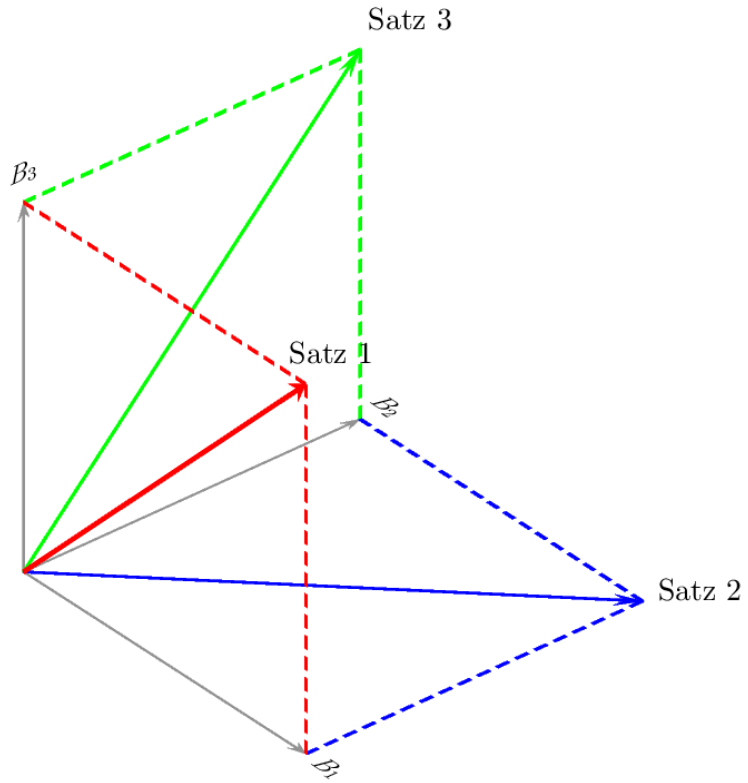


Abbildung 2.1: Vektorvisualisierung

Somit kann man feststellen, dass diese 3 Vektoren einen Abstand von jeweils $\frac{\pi}{3}$ zueinander haben. Die Sätze sind somit vergleichbar ähnlich. Dieser Ansatz hat also Potential, um ein vielversprechendes Resultat zu liefern. Jedoch sind noch 2 Faktoren zu überprüfen, welche Einfluss auf das Resultat haben könnten.

- **Vektorlänge:** Die Vektorlänge (Betrag [KN02]) wird im obigen Beispiel durch die Anzahl Wörter bestimmt. Je länger der Text, desto mehr Wörter enthält er. Die Anzahl Wörter hat einen direkten Einfluss auf die Länge des Vektors. Es wird jedoch der Winkel ϕ zwischen den Vektoren zur Bestimmung der Ähnlichkeit verwendet. Da das Skalarprodukt die

Vektoren normalisiert, hat die Textlänge somit keinen Einfluss auf das Resultat! Werden aber die Vektoren bewusst normalisiert, so ist das Skalarprodukt ohne die Normalisierung einfacher zu berechnen.

- **Normierung der Häufigkeit:** Da in vielen Sprachen Artikel viel häufiger vorkommen als ein bestimmtes Nomen, beeinflussen diese die Richtung unseres Vektors überdurchschnittlich stark. Man könnte nun alle 'Füllwörter', die wenig Informationsgehalt übertragen einfach entfernen.
  Eine bessere Idee wäre es die Wörter nach Häufigkeit zu gewichten! Somit wird ein seltenes Wort stärker gewichtet, als welche die viel häufiger vorkommen.

Der Wunsch nach einem Lösungsmodel, welches auf die semantische Analyse verzichtet, legte uns die probabilistischen Konzepte nahe. Herr Prof Dr. Andreas Müller hat mit einigen Hinweisen und dem 'Wahrscheinlichkeit und Statistik'-Skript [Mü15] einige Hinweise geliefert die auf eine erfolgsführende Lösung zeigten. Jedoch durfte nicht vergessen werden, dass die Implementierung in Haskell schon genug Hürden in sich bringt, daher wurde versucht eine möglichst einfache Lösung zu wählen. Grundsätzliche Überlegungen sollten so früh wie möglich gemacht werden, jedoch ist ein 'minimal viable product' sehr erstrebenswert, daher wurden sämtliche Optimierungen auf einen späteren Zeitpunkt verschoben.

2.2.3 Probabilistisches Lösungsmodell

Durch das Weglassen der kontextbezogenen Verarbeitung wurde entschieden den Schwerpunkt auf die probabilistischen Konzepte zu setzten. Da vermutlich nicht ein einfach umsetzbares Rezept gefunden werden konnte, wurde zuerst ausgiebig nach Ideen gesucht. Die dokumentierten Ideen können später noch detailierter getestet werden.

Bayes-Ansatz

Inspiriert von 'A Plan for Spam' [Gra15] und dem Wahrscheinlichkeit und Statistik Skript, erschien der Bayes Ansatz als sehr geeignnet. Jedoch ist nicht die Frage ob 'Spam' erhalten wurde oder nicht, sondern viel eher wie wichtig das Wort im Zusammenhang mit dem Text ist. Daraus resultierte folgende Überlegung:

$P(T)$	Wahrscheinlichkeit diesen Text auszuwählen.
$P(W)$	Wahrscheinlichkeit, das Wort (W) in allen Texten zu finden.
$P(W\|T)$	Wahrscheinlichkeit, dass der ausgwählte Text das Wort (W) enthält.

Nun muss $P(T|W)$ ermittelt werden. Diese Wahrscheinlichkeit sagt aus, wenn ein bestimmtes Wort gemeint ist, wie gross die Wahrscheinlichkeit ist, dass genau dieser Text gesucht ist.

$$P(T|W) = \frac{P(W|T)P(T)}{P(W)}. \tag{2.1}$$

Die Formel (5.1) entspricht dem Satz von Bayes.[Mü15]. Es ist unschwer zu erkennen, dass Wörter die selten vorkommen, aber im einem Text regelmässig erwähnt werden einen stärkeren Einfluss auf das Resultat haben.

Informationsentropie

Aus der Informatoins und Codierungstheorie ist bekannt, dass Zeichen die öfter vorkommen, weniger Einfluss auf den Informationsgehalt von Kommunikation haben. Diese Aussage wurde von Claude E. Shannon mit der folgender Gleichung postuliert:

$$I(p) = -\log_2 p \tag{2.2}$$

Für p müssen die Randbedinung $0 <= p <= 1$ eingesetzt werden. Analog kann somit für Texte ein Wort genommen werden und dessen Informationsgehalt anhand der Formel nach Shannon ermittelt werden. Ein Wort kann höchstens mit der Wahrscheinlichkeit 1 auftretten und mindestens 0. Somit lässt sich diese Aussage mit folgender Grafik veranschaulichen. ->FANCY GRAPHIC 2<- Wie unschwer zu erkennen ist, werden somit Wörter die häufig vorkommen, sehr niedrige Werte erhalten und somit die Richtung des Vektors nur wenig beeinflussen. Sollte aber ein Wort nicht häufig im Text vorkommen, so hat es einen deutlichen Einfluss auf das Ergebnis.
Problematisch ist an dieser Stelle, dass Wörter die oft vorkommen in spezifischen Texten vermutlich auch sehr nah mit dem Text gekoppelt sind. Zur Veranschaulichung kann eine Passage aus einem Tennisartikel genommen werden. Handelt sich der Artikel um Roger Federer, wird diese Wortkombination vermutlich öfter vorkommen, als Stanislas Wawrinka. Trotzdem kann es sein, dass Stanislas Wawrinka noch eine kurze Erwähnung hat, er könnte ja am kommenden Tag einen Match austragen. Das Problem liegt nun darin, dass 'Stansilas' als Wort mehr Einfluss auf die Vektorrichtung hat, als 'Roger'. Somit wäre der 'Roger Federer'-Artikel einfacher zu finden sein, wenn eine spezifische Suche nach 'Stanislas' vollzogen wurde. Als Konsequenz würde man viele falsch-positive Varianten erhalten mit dieser Methode.
Diese Methode hat sicherlich potential, muss aber in Kombiation mit einem naive Bayes verfahren angewendet werden. Oder es müssen andere Mechanismen greifen, um die relative Häufigkeit von spezifischen Wörtern in Texten zu bewerten.

Skalarvergleich

Der naheliegendste Vergleich von ähnlichen Texten basiert auf dem Skalarprodukt. Bei dieser Methode wird ausgenutzt, dass alle Vektoren normiert auf einer n-Dimensionalen Einheitskugel liegen. Wir können somit die Ähnlichkeit zweier Vektoren mit dem Skalarprodukt wie folgt ausdrücken:

$$\cos(\varphi) = \frac{\langle x, y\rangle}{\sqrt{\langle x, x\rangle}\,\sqrt{\langle y, y\rangle}} = \frac{x_1 y_1 + \cdots + x_n y_n}{\sqrt{x_1^2 + \cdots + x_n^2}\,\sqrt{y_1^2 + \cdots + y_n^2}} \tag{2.3}$$

Wenn darauf geachtet wird, dass die Vektoren $\vec{x}$ und $\vec{y}$ bereits normiert sind, so entspricht das der Gleichung

$$\cos(\varphi) = \frac{\langle x, y\rangle}{\sqrt{1}\,\sqrt{1}} = \langle x, y\rangle \tag{2.4}$$

Bei normierten Vektoren kann somit einfach das Skalarprodukt verwendet werden. Eine weitere Überlegung könnte an diesem Punkt noch gemacht werden. Der Cosinus kann komplett weggelassen werden. Die Ähnlichkeit kann mit einem 'Identitätskoeffizient' i ausgedrückt werden.

$$i = \langle x, y\rangle = x_1 y_1 + \cdots + x_n y_n \tag{2.5}$$

Somit steht fest, dass wenn i nah bei 1 liegt, der Artikel sehr ähnlich ist, wenn i gegen 0 geht, sind die zwei Texte nicht ähnlich.

Hyptothesentest

Eine weitere Methode zum Vergleich zweier Texte könnte im Hypothesentest verborgen sein. Der Hypothesentest[1] hat den Ursprung in der mathematischen Statistik. Es kann eine Hypothese aufgestellt werden und diese dann mit Hilfe des Tests angenommen oder verworfen werden. In dem konkret vorliegendenden Fall entspricht die Nullhypothese H_0 der Vermutung, dass zwei Artikel ähnlich sind. Die Alternativhyptohese H_1 wäre somit die Behauptung, dass die Artikel nicht gleich (unterschiedlich) sind. Das Resultat des Tests könnte somit Aufschluss über die Ähnlichkeit zweier oder mehrerer Texte liefern. Durch setzten des Signifikanzniveaus α könnte entschieden werden, ab welchem Schwellwert, die Texte zu 'unterschiedlich' sind.
Bereits beim konkreten Ausarbeiten eines Beispiels wurde festgestellt, dass bei Verwendung diskreter Zahlen nicht geeignet sind. Lange Artikel haben dadurch höhere Werte als kurze Artikel. (siehe Vektorlänge 5.2.2ff). Die Werte müssen aus diesem Grund normiert werden. Aus diesem Grund sind die Werte zu jedem Zeitpuntk < 1. Dies kann auch gezeigt werden, weil der Artikel auf sich selbst abgebildet als Ergebnis die Länge 1 liefert. Aus dieser Feststellung folgt, dass sowohl der χ^2-Test als auch der G-Test ungeignet sind. Der G-Test funktionert bei kleinen diskreten Werten zwar besser als der chi^2, jedoch werden bei beiden Tests Wertebereiche ab drei respektive fünf empfohlen.
Anhand dieser Erkenntnisse wurde beschlossen, das Hyopthesentestmodel nicht weiter zu verfolgen.

[1] Hypothesentest: `http://de.wikipedia.org/wiki/Statistischer_Test`

3 Erarbeitung des Konzepts

3.1 Einleitung

Da die Studienarbeit gewissermassen eine Realisierbarkeitsstudie und komplettes Neuland umfasst, wurde relativ früh Wert auf eine 'realistische Zielsetzung' gelegt. Zur Problemstellung gehören nicht nur neue Programmierkonzepte, aber auch Neuland im Bereich von "Data Miningünd "Data Classification". Da Risiken nur gut berechenbar sind, wenn diese bekannt sind, war ein Anliegen, diese in einer frühen Projektphase bereits auszuschliessen. Es wurde somit Wert gelegt auf das Konzept einer erfolgreichen Lösungsfindung.

3.2 Risikoanalyse

3.2.1 Analyse

Die Risikoanalyse diente als Aufhängepunkt für die komplette weitere Planung und ist somit ein wichtiger Bestandteil für das weitere Vorgehen. Anhand der Riskoanalyse und unserem Vorgehen (gem. Technischer Bericht seite xyz) können die nächsten Schritte geplant werden.

	Name	P [1]	C [2]	Beschreibung
	Benutzer			
R-B1	GUI Komplexität	2	5	Das User Interface ist nicht intuitiv zu bedienen, der Benutzer fühlt sich bei der Verwendung überfordert.
R-B2	Produkt	1	3	Technologie ist eine spannende Sache, aber es soll nicht vergessen werden, dass ein Produkt aus der Arbeit entstehen soll!
	Anforderungen			
R-A1	Umsetzung	1	3	Die Vision ist bekannt, aber der Weg nicht. Es besteht ein Risiko sich zu verlieren. Das Ziel kann durch die technologische Last schnell aus den Augen verloren werden.
R-A2	Produktvorstellung	3	5	Die Teammitglieder und der Betreuer haben unterschiedliche Vorstellungen über das finale Produkt.
R-A3	Aufgabenstellung	1	1	Die Aufgabenstellung kann zu hoch gegriffen sein und es ist unmöglich innerhalb der gegebenen Zeit diese zu lösen.
	Komplexität			
R-K1	Framework	4	5	Durch mangelnde Haskell Erfahrung kann es zu Fehlbeurteilungen bei der Wahl der Frameworks kommen, die im Nachhinein sehr aufwendig zu korrigieren sein werden.
R-K2	Haskell Grundlagen	4	5	Haskell ist nicht innerhalb nützlicher Frist zu lernen.
R-K3	Textanalyse ungenau	4	5	Es ist möglich, dass die Textanalyse keine genauen Resultate liefert.
R-K4	Performance Probleme	3	3	Die Suche funktioniert zur vollsten Zufriedenheit, die Geschwindigkeit lässt aber zu wünschen übrig.
R-K5	Haskell lazyness	2	5	Durch die 'lazy evaluation' von Haskell kann es zu Problemen kommen, die noch nicht angetroffen wurden.

[1] Probability (Eintrittswahrscheinlichkeit)
[2] Cost (Kosten im Eintrittsfall)

R-K6	Datenmenge	4	2	Die Datenmenge wird ein problematisches Ausmass annehmen. (Wikipedia ist $\tilde{5}$0GB)
	Organisation			
R-D1	Dokumentation	2	3	Die Dokumentation wird zu Gunsten der Entwicklung vernachlässigt und wird nur flüchtig geführt.
R-D2	Overhead	4	1	Die Studienarbeit umfasst eine Entwicklungszeit von 240h. Nicht zu vernachlässigen sind Termine, Dokumentationen, etc. Die Nebenschauplätze sollen nicht vergessen werden!
R-D3	Entwicklungsumgebung	2	3	Es kann für Haskell keine vernünftige IDE gefunden werden. Konsequenzen davon wären eine ineffiziente Programmierung.
R-D4	Einarbeitung	2	4	Eine nur schleppende Einarbeitung in die Aufgabenstellung und die technologischen Hürden von Haskell.
	Team			
R-T1	Freiheiten	1	5	Durch die sehr offene Aufgabenstellung besteht ein Risiko, dass das Team mit den Freiheiten nicht umgehen kann.
R-T2	Motivation	1	5	Durch die komplexe und umfangreiche Aufgabenstellung kann man leicht die Sicht auf das Ganze verlieren. Überforderung und Frustration können sich negativ auf die Motivation des Teams auswirken.
	Planung & Kontrolle			
R-P1	Falsche Prioritäten	3	3	Es kann augrund von technologischen Problemen und der Komplexität der Aufgabenstellung zu Fehlentscheidungen bezüglich der Prioritäten gewisser Massnahmen kommen.
R-P2	Zeitmanagement	2	5	Beide Teammitglieder sind Teilzeitstudenten. Es kann sehr einfach durch geschäftliche Prioriäten zu Zeitengpässen kommen.
R-P3	Sackgassen	4	3	Technologische, konzeptionelle Sackgassen können das Projekt gefährden.

3.2.2 Konsequenzen

Die Konsequenzen der Risikoanalyse sind sehr aussagekräftig. Es wurde schon am Anfang vermutet, dass die Komplexität der Arbeit die Hauptgefahren birgt. Dies kann nun bestätigt werden. Die Risiken mit dem grössten Handlungsbedarf, befinden sich hauptsächlich in diesem Teil und sollten somit etwas genauer bewertet werden.

		Schadenausmass				
		1	2	3	4	5
Eintrittswahrscheinlichkeit	5					
	4	R-D2	R-K6	R-P3		R-K1 R-K2 R-K3
	3	R-A1		R-K4 R-P2		R-A2
	2	R-B1		R-D1 R-D3	R-D4	R-K5 R-P2
	1	R-A3		R-B2		R-T1 R-T2

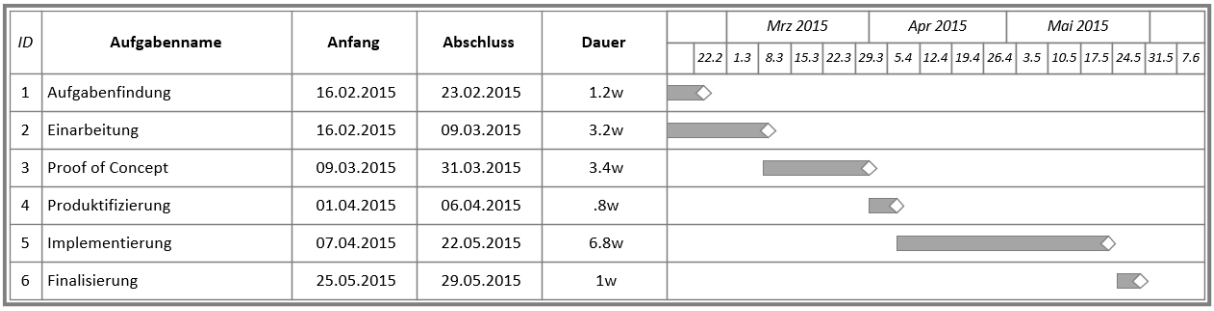
Abbildung 3.1: Risiko Matrix

- **R-K2, R-K3:**
  Diese Punkte können beseitig werden mittels guter **Einarbeitung**.
- **R-K1:**
  Die beste Massnahme sich in ein neues Gebiet einzuarbeiten, ist Sachen auszuprobieren. Somit ist die effektiveste Massnahme ein **Proof of Concept**.
- **R-A2:**
  Eine Herausforderung in sich ist die **Erarbeitung des Konzepts**, durch gezieltes Abwägen und Erfahrung sollten Risiken in diesem Bereich gut erkennbar sein.

Auf andere Punkte wird hier nicht näher eingegangen, da viele mit gutem Projektmanagement automatisch beseitigt werden können. Andere (R-P2) können mit Planung minimiert werden.

3.3 Meilensteinplanung

Um in dem engen Zeitraum von knapp 16 Wochen die nötigen Meilensteine zu setzten, ist ein grober Projektplan notwendig. Dieser soll nicht einen definitiven Charakter haben, er soll lediglich dazu dienen um aufzuzeigen welche Phase der Entwicklung gerade in Arbeit ist. Durch das Proof of Concept wird es nötig sich Gedanken zu machen, in welchem Umfang ein "Wegwerf-Produkt gebaut wird.



ID	Aufgabenname	Anfang	Abschluss	Dauer
1	Aufgabenfindung	16.02.2015	23.02.2015	1.2w
2	Einarbeitung	16.02.2015	09.03.2015	3.2w
3	Proof of Concept	09.03.2015	31.03.2015	3.4w
4	Produktifizierung	01.04.2015	06.04.2015	.8w
5	Implementierung	07.04.2015	22.05.2015	6.8w
6	Finalisierung	25.05.2015	29.05.2015	1w

Abbildung 3.2: Meilensteinplanung

Der Meilensteinplan ist grundsätzlich gemäss unserer Vorgehensweise gegliedert. (Siehe technischer Bericht). Die Meilensteine umfassenden folgenden Umfang und sollten umbedingt eingehalten werden können:

- **Ende des Proof of Concept:**
  - Datum: 31.03.2015
  - Ziel: Ende des Proof of Concept, die Realisierbarkeit ist bewiesen.
  - Erfüllungskriterium: Proof of Concept kann die einzelnen Module erfüllen - es ist *kein* Architekturdurchstich nötig.
- **Ende der Implementierung:**
  - Datum: 22.05.2015
  - Ziel: Nach diesem Zeitpunkt wird kein Feature mehr implementiert - das Produkt steht.
  - Erfüllungskriterium: Das Produkt steht, kann noch kleine Bugs haben, aber die Features sind klar und ersichtlich.
- **Abgabe**
  - Datum: 29.05.2015
  - Ziel: Arbeit ist abgegeben, Produkt ist funktionsfähig und online.
  - Erfüllungskriterium: Arbeit abgegeben.

3.4 Architekturgedanke

Da die Anforderung an ein Proof of Concept nun gestellt ist, ist es wichtig sich Gedanken zur Architektur zu machen. In Anbetracht der Problemstellung ist eine Architektur mit klaren Einelzkomponenten sinnvoll. Die Datenbank dient somit als zentraler Aufhängepunkt. Dies ermöglicht sehr einfache und kleine Komponenten zu entwickeln und im Problemfall Java einzusetzten ohne grosse Integrationsprobleme. Im Grunde soll die gesamte Arbeit in Haskell geschrieben werden, jedoch muss in Erwägung gezogen werden, während des Proof of Concepts für gewisse Komponenten Java zu verwenden. Haskell soll sicher bereits früh verwendet werden um die Framework Evaluation zu untermauern und Risiken bereits während des Proof of Concepts auszuschliessen. Diese treibenden Faktoren haben am Schluss zu folgendem Architekturgedanken geführt.

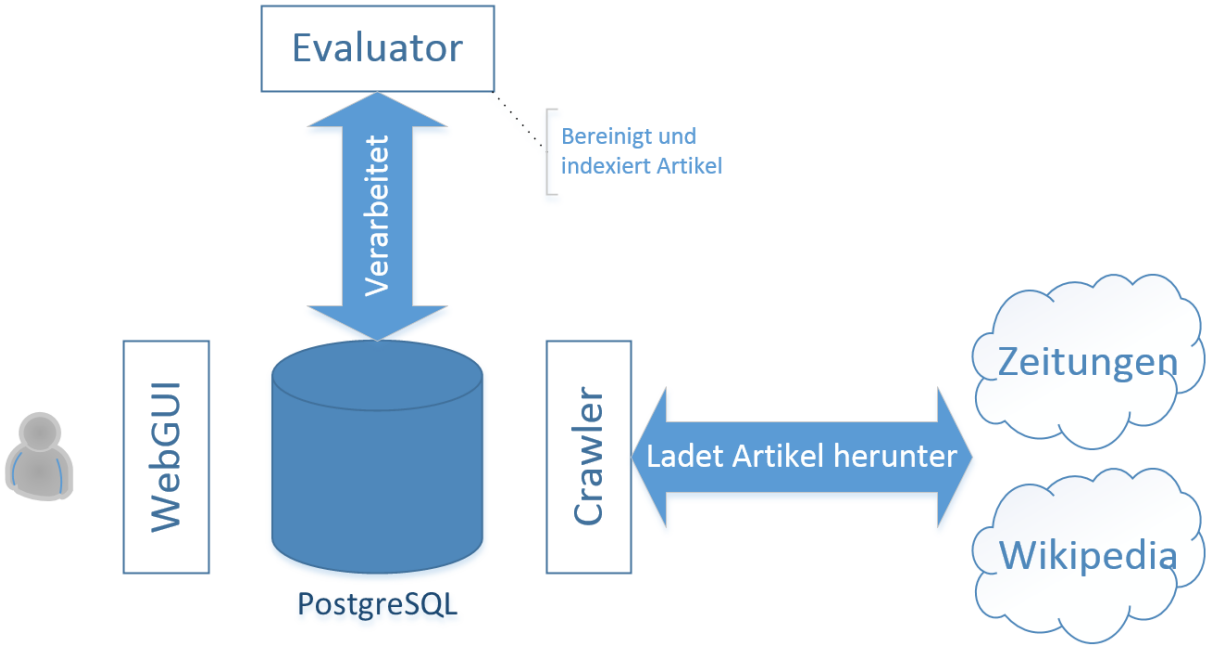


Abbildung 3.3: Konzeptskizze der Architektur

4 Proof of Concept

4.1 Einführung

Durch die Risikoanalyse wurde das Proof of Concept motiviert. Das Proof of Concept sollte nicht die ganze Aufgabenstellung abdecken oder einen technologischen Durchstich bringen. Vielmehr war es wichtig, dass die Schwierigkeit von technologischem Neuland und komplexer Aufgabenstellung auf das Kernproblem reduziert werden konnte. Es wurde versucht die thematische Problemstellung von der technologischen zu trennen. Das Proof of Concept sollte primär zeigen, ob die Aufgabenstellung lösbar ist. Durch die Architekturüberlegungen aus der Konzeptionsphase ist es möglich die unterschiedlichen Komponenten in verschiedenen Technologien zu entwickeln.

4.2 Grobarchitektur

In der Literatur ist der Einsatz von UML sehr geläufig, um Architekturen und Komponenten aufzuzeichnen. Es wurde bewusst dagegen entschieden, da im Proof of Concept die Problemstellung zuerst zerlegt werden musste. Es wurde daher bewusst eine sehr einfache Darstellung gewählt. Die Aufgabe sollte zuerst auf die grossen Felder aufgeteilt werden, um zu entscheiden, welche ein Risiko bringen und was in einem Proof of Concept wirklich umgesetzt werden sollte. Aus der Konzeptionsphase haben sich die drei grossen Komponeten als eine gute Aufteilung erweisen. Alle drei Bausteine verwenden die Datenbank als zentrale Drehscheibe und speichern oder lesen Daten. Die einzelnen Teilkomponenten wurden wie folgt gegliedert:

- **Crawler:** Der Crawler hat die zentrale Aufgabe diverse Quellen abzusuchen und Texte in die Datenbank zu laden. Im Gegensatz zu einem klassischen Webcrawler werden RSS Feeds von Zeitungen und Newsseiten als Grundlage ausgwählt. Die RSS Feeds dienen dazu in einem weiteren Schritt die Informationen aus der Seite zu extrahieren und so an die wichtigen Informationen zu kommen.
- **Evaluator:** Der zentrale Kern des Projektes ist der Evaluator. Dieser soll die vorhandenen Daten bewerten und semantische Vergleiche zu Dokumenten ermöglichen. Der Evaluator wird die Kernkomponente bilden und ist nach erster Analyse sowohl technologisch als auch konzeptionell die komplexeste Komponente.
- **Supplier:** Auf der Benutzerseite soll eine Webseite aufgerufen werden können, die die wichtigen Informationen anzeigt und die Daten aufbereitet. Ziel ist es hier nicht noch durch eine spezifische REST Schnittstelle das Projekt komplexer zu gestalten oder aufzublasen. Daher wird die ganze Komponente Supplier gennant. Ursprünglich war der Gedanke ein WebGUI und eine REST Schnittstelle in Haskell anzubieten.

Aus oben erwähnten Gründen wurde entschieden, dass die Komponenten Supplier und Crawler einen guten Einstieg in die Haskell Welt ermöglichen. So konnte diese Phase der Arbeit auch genutzt werden um Frameworktechnologien zu evaluieren. Für die Komponente Evaluator wurde Java als Technologie eingesetzt. Durch den Einsatz von Java sollte es möglich sein sich stark auf die Aufgabenstellung zu konzentrieren und nicht durch technologisches Verständnis ausgebremst zu werden. Bewusst wurde auf einen kompletten architektonischen Durchstich verzichtet, da das Risiko von nicht harmonierenden Komponenten zu diesem Zeitpunkt als sehr gering eingeschätzt wurde. Die Kernaufgabe des Proof of Concept liegt bei der Risikominimierung. Daher liegt der Schwerpunkt dieser Phase auf dem Fund der richtigen Haskell Frameworks und dem Verständnis der Thematik.

4.3 Crawler

Für den Proof of Concept wurde 'The Guardian' als Quelle ausgewählt. Die Wahl fiel ziemlich einfach aus, da 'The Guardian' als einer der wenigen Verlagshäuser ein API bereitstellt, um auf die aktuellsten Artikel zugreifen zu können. Neben den Artikeln wurde auch entschieden im Proof Of Concept die Wikipedia Daten einzuspielen. Ziel war es ein API in Haskell anzubinden und mit grösseren Datenmengen Erfahrungen zu sammeln.

4.3.1 The Guardian

Obwohl 'The Guardian' ein REST-ähnliches API anbietet [1], war es nicht einfach die Artikel in die Datenbank zu importieren. Es stellten sich zunächst mehrere Probleme:

- Die API stellt nur die Links zu den entsprechenden Artikel zur Verfügung (ähnlich einer RSS-Feed). Der Inhalt des Artikels musste separat nachgeholt werden.
- Der Artikelinhalt enthält Multimedia-Inhalte, wie Videos oder Bild. Diese müssen nachträglich gesäubert werden, um nur den Text zu bekommen.
- Die API lieferte nicht nur Artikle, sondern auch andere Inhalte, wie Life-Feeds zu einem Spiel oder Videodokumentation.

Es wurde versucht die unnötigen Inhalte zu säubern. Jedoch stellte sich dies als schwierig heraus. Oft war es nötig Bilder oder Videos im HTML Code auszuwerten, um diese zu löschen. Da nicht nur Gaurdian zusätzliche Inhalte im Text mitlieferte und dies potenziell auch bei anderen Verlagshäuseren verbreitet war, musste eine Alternative gefunden werden, wie der reine Artikelinhalt "generischäus dem Artikel extrahiert werden konnte. Es musste eine Lösung gefunden werden, die auch für andere Anbieter funktioniert und wenig Implentierungsaufwand mit sich brachte. Bilder, Videos und Text zu trennen, ist durch verarbeiten des HTML Inhalts möglich. Diese Lösung bringt den Nachteil mit sich, dass oft auch die Bildbeschreibung als Text wahrgenommen wurde. Es musste somit ein besserer Lösungsansatz gefunden werden.

[1] `http://open-platform.theguardian.com/`, 12.04.15

Abbildung 4.1: The Guardian Artikel

Readability Service

Die Lösung des Problems brachte Readability [2]. Bei Readability handelt es sich um ein eine App, welche Artikel einheitlich darstellt, um diese besser lesbar zu machen. Unter anderem werden die Artikel formatiert und unnötige Inhalte, wie etwa Galerien oder Videos entfernt. Zusätzlich werden auch Artikel welche über mehrere Seiten verteilt sind auf einer Seite ausgegeben.
Das Säubern von Artikeln wird von Readabiltiy auch als API angeboten. Es kann somit aus der Applikation als einfacher Webservice genutzt werden. Nach eingen Tests mit medienbeladenen Artikeln entschieden auf diesen Service zu setzten. Die Ergebnisse waren konstant gut und konnten somit verwendet werden. Ein weiterer Vorteil dieser Lösung war das einfache JSON API, welches in Haskell sehr einfach angesprochen werden konnte:

[2] `http://readability.com/`, 12.04.15

```haskell
buildUrl :: ReadabilityConfig -> URL -> URL
buildUrl config articleUrl = S.convertUrlOrFail $ apiUrl ++ token ++
 ↪  "&url=" ++ articleLink
    where
      apiUrl = "https://readability.com/api/content/v1/parser?token="
      token = apiKey config
      articleLink = exportURL articleUrl

callApi :: ReadabilityConfig -> URL -> IO (Maybe ReadabiltiyResonse)
callApi config urlToArticle = do
        responseBody <- S.sendRequest $ buildUrl config urlToArticle
        return $ decode responseBody

callApiDefault :: URL -> IO (Maybe ReadabiltiyResonse)
callApiDefault urlToArticle = do
                responseBody <- S.sendRequest $ buildUrl defaultApiConfig
                                urlToArticle
                return $ decode responseBody

extractContent :: Maybe ReadabiltiyResonse -> String
extractContent a = case a of
                Just value -> content value
                Nothing    -> "failed"
```

Das Problem bestand nun darin, dass Readability nicht direkt den Textinhalt lieferte, sondern diesen noch in HTML formatierte. Die HTML Formatierung entsprach einem einheitlichem Format. Aus diesem Grund konnte das Problem relativ einfach mittels der Pandoc[3] Bibliothek gelöst werden. Mit Pandoc wurde der HTML Code zu einfachem Text konvertiert.

4.3.2 Wikipedia Daten

Neben 'The Guardian' wurde entschieden auch alle Wikipedia Artikel in die Datenbank zu importieren. Wikipedia stellt JSON oder XML Dumps zu Verfügung.[4]. Diese Dumps bieten sich an um an die Informationen der Artikel auf einfachem Wege zu gelangen. Es werden verschiedene Dump-Files angeboten, das für uns interessanteste Packet beinhlatet alle Artikel ohne Meta-Daten und Versionshistorie. Die Daten kommen komprimiert als 10GB grosse Datei, welche entpackt 49GB belegen. Nach kurzem Zählen wurde die Zahl der Artikel auf 15'438'350 geschätzt. Da der ganze Import den Umfang des PoC sprengen würde, wird im folgenden nur versucht ein Teil zu importieren. Neben dem ganzen Dump, stellt Wikipedia auch den Dump in mehreren Teilen zur verfügung. Im folgenden wird versucht den Ersten davon zu importieren, welcher ingesamnt 6280 Artikel enthält.

[3] `http://johnmacfarlane.net/pandoc/`, 14.04.15
[4] `http://en.wikipedia.org/wiki/Wikipedia:Database_download`, 12.04.15

Datenbankimport

Als erstes wurde versucht, die Wikipedia Daten in die Daten zu importieren, denn die XML Dumps entsprechen der Wikipedia Datenstruktur. Wikipedia selbst stellt einen PHP-Importer zur Verfügung[5]. Dieser naive Ansatz hat den Vorteil, dass die frischen Wikipedia-Daten jederzeit in der Datenbank sind und die relevanten Daten sehr einfach in die gewünschte Tabellenstruktur kopiert werden können.
Beim Versuch die Daten in die Datenbank zu importieren stellten sich mehrere Probleme:

- Das Programm zum Importieren wurde für MySQL entwickelt und wird aber mit PostgreSQL verwendet. Es wurden zahlreiche Syntax-Probleme festgestellt. Anfänglich wurde versucht diese von Hand in den Griff zu bekommen, schlussendlich würde der ganze Aufwand aber viel zu gross werden.

- Wie der folgende Auszug zeigt, werden die Artikel in der Datenbank im Wikimedia-Syntax abgespeichert. Diese müssen später nochmals prozessiert werden, um zum Text-Inhalt zu gelangen. Ein Beispiel sieht wie folgt aus:

  ```
  == Eigenschaften ==
  Insgesamt darf eine .io-Domain zwischen drei und 63 Stellen lang sein und nur
  [[alphanumerische Zeichen]] beinhalten.<ref>{{Internetquelle | url =
  https://www.united-domains.de/io-domain | titel = Eigenschaften einer .io-Domain
  | hrsg = [[United-Domains|united-domains]] | zugriff = 2013-12-22}}</ref> Jede
  [[natürliche Person|natürliche]] oder [[juristische Person]] darf eine
  .io-Domain registrieren, ein lokaler Wohnsitz oder eine Niederlassung sind nicht
  erforderlich. Nur Adressen unterhalb einer [[Second-Level-Domain]] wie zum ...
  ```

Haksell-Importer

Nach langer Recherche wurde entschieden, dass die beste Alternativ ein Haskell Programm darstellt, welches die Daten importiert. Neben dem anfänglich zusätzlichem Aufwand, bringt es viele Vorteile mit sich. Denn bei der Verwendung von einem fremden Importer muss dessen Datenbankschema übernommen werden. Dieser würde Daten importieren welche schlussendlich gar nicht gebraucht werden. Bei der eigenen Lösung, kann genau kontrolliert werden, welche Daten übernommen werden. Ausserdem lässt sich der eigene Importer bei Problemen mit den Daten schneller korrigeren und auch bei Performanceproblemen schneller anpassen.
Für das Parsen der XML-Datei wurde ein Library verwendet, welches nicht zu stark auf Abstraktionen wie Monad oder Arrows setzt. Stattdessen wird die XML-Struktur als eine Liste von Tags angesehen, auf dessen mittels 'filter' und 'partitions' die relevanten Tags extrahiert werden können.

Der folgende Abschnitt zeigt eine vereinfachte Struktur der Wikipedia Daten und den passenden Haskel Code, um den Inhalt eines Artikels zu extrahieren.

[5] `http://meta.wikimedia.org/wiki/Data_dumps/Import_examples`, 24.05.15

```
-- <mediawiki>
--     <page>
--       <title></title>
--       <revision>
--           <text></text>
--           <timestamp></timestamp>
--       </revision>
--     </page>
-- </mediawiki>

-- firstRevision is the representation of Tag <revision>
takeContentFromRevision firstRevision = takeContent "text" firstRevision

-- from left to right: drop till tagName found, drop that <tagName> and
 ↪  then take the head
-- which is the inner text of <tagName>
takeContent tagName tagList = fromTagText $ head $ drop 1 $ dropWhile
 ↪  (~/= openingTag tagName ) tagList
```

4.4 Evaluator

4.4.1 Einleitung

Der Evaluator bildet das zentrale Glied der Arbeit und soll die Artikel bewerten. Bewusst wurde Java als Programmiersprache für den Prototypen eingesetzt um die Entwicklung zu Beschleunigen. Der Fokus sollte ganz klar auf dem Algorithmus liegen und nicht der verwendeten Programmiersprache.

4.4.2 Rahmenbedinungen

Die Rahmenbedinungen für den Prototypen waren sehr einfach. Es wurde komplett in **Java 8** programmiert, als Bibliotheken wurde lediglich **Junit4** und die aktuelle Version von **jBlas** verwendet. JBlas war prädestiniert für die Berechnung von Vektoren und allen mathematischen Operationen. Da es sich um einen Prototyp handelte, war das Testing nicht von zentraller Bedeutung. Jedoch war es wichtig, gewisse Edge-Cases zu definieren um die Funktionsweise des Algorithmus zu garantieren. Aus der Einarbeitung haben wir festgestellt, dass die Mathematik hinter dem Algorithmus stimmen muss, daher wurden folgende Rahmenpunkte festgelegt:

- **Normierung**
  Jeder Vektor hat die Länge 1. Unabhänig von der Länge des Strings, muss der Vektor eine konstante Länge auffweisen.

- **Skalarprodukt** $\vec{a} \cdot \vec{a} = 1$
  Das Skalarprodukt zweier identischer Vektoren ist 1.

Diese Rahmenbedinungen haben geholfen den Code zu stabilisieren. Danach konnte der Fokus auf die Daten und deren Verhalten gelegt werden.

4.4.3 Vergleich der verschiedenen Lösungsansätze

Aus der Einarbeitungsphase haben sich eigentlich zwei Lösungen auskristalisiert, die etwas näher untersucht werden sollten. Einerseits ist der Bayes-Klassifikator eine Lösung, andererseits steht noch der Informationsgehalt nach Claude Shannon auf dem Prüfstand. Da das Lösungsmodel mit dem Informationsgehalt vermutlich nicht als eigenständige Lösung zu verwenden ist, wurden folgende 2 Varianten ausprogrammiert:

- **Bayes-Klassifikator**
  Die Lösung soll einen Vektor bilden Mittels der Werte des Bayes-Klassifikators
- **Bayes-Klassifikator nach Informationsgehalt**
  Gemäss der Aussage von Claude Shannon wird die Formel nach erfolgreicher Bayes-Klassifizierung auf den Vektor angewendet. Dies bewirkt eine Korrektur der Werte die Nahe an den Extremalwerten von 0 und 1 liegen.

Nach zahlreichen Versuchen und der Feststellung, dass die Ergebnisse sehr ähnlich sind, wurde versucht die Problemstellung besser festzuhalten.

4.4.4 Benchmark

Ein weiterer Punkt, der aus dem Proof of Concept zu beantworten war, war die Qualität des 'Evaluators'. Wie gut arbeitet der Algorithmus? An welchen Stellen existiert noch Potential um die Resultate zu verbessern? Als logische Konsequenz musste ein Benchmark entwickelt werden, der sehr genau Aussagen über die Qualität machen konnte. An dieser Stelle wurde bewusst auf einen computerbasierten Vergleich verzichtet. Ein sehr zuverlässiges Gefühl für Inhalte von Texten hat nach wie vor der Mensch, daher wurde der Algorithmus auf diesen Qualitäten aufgebaut. Es wurde beschlossen, 15 Artikel mit unterschiedlicher Länge und Gliedbarkeiten in drei Kategorien zu verwenden. Dabei sollten drei Artikeln jeweils sehr nah beieinander liegen und zwei eher grob ins Thema passen. Nach intensiver Recherche [6] wurden folgende 15 Artikel ausgewählt und wie folgt klassifiziert:

[6] Stand der Artikel gem. 22. März 2015

	Kategorie: **Sport**
01	http://www.theguardian.com/sport/2015/mar/20/roger-federer-tomas-berdych-indian-wells-semi-final
02	http://www.nytimes.com/2015/03/21/sports/tennis/roger-federer-and-milos-raonic-win.html?ref=tennis
03	http://www.usatoday.com/story/sports/tennis/2015/03/20/federer-rolls-past-berdych-to-reach-indian-wells-semifinals/25100785/
04	http://www.telegraph.co.uk/sport/tennis/australianopen/11364410/Roger-Federer-crashes-out-of-Australian-Open-after-shock-defeat-to-Andreas-Seppi-in-third-round.html
05	http://www.bbc.com/sport/0/tennis/30071708
	Kategorie: **Politik**
06	http://www.theguardian.com/commentisfree/2015/mar/01/guardian-view-boris-nemtsov-watershed-russia
07	http://www.nytimes.com/2015/03/09/world/europe/suspect-in-russian-politicians-killing-blows-himself-up-report-says.html
08	http://www.bbc.com/news/world-europe-31693234
09	http://www.theguardian.com/world/2014/nov/19/new-cold-war-back-to-bad-old-days-russia-west-putin-ukraine
10	http://www.nytimes.com/2015/02/09/world/crisis-in-ukraine-underscores-opposing-lessons-of-cold-war.html
	Kategorie: **Technologie**
11	http://www.theguardian.com/technology/2015/mar/20/google-illegally-took-content-from-amazon-yelp-tripadvisor-ftc-report
12	http://www.geekwire.com/2015/report-alleges-google-took-product-rankings-from-amazon-to-boost-its-search-results/
13	http://uk.businessinsider.com/what-the-new-ftc-report-says-about-google-2015-3?r=US
14	http://www.theguardian.com/technology/2014/may/28/google-self-driving-car-how-does-it-work
15	http://www.nytimes.com/2014/05/28/technology/googles-next-phase-in-driverless-cars-no-brakes-or-steering-wheel.html

Anschliessend wurden Texte ausgewählt die gut in eine Kategorie passen. Es wurden Einleitungstexte von Artikeln anderer Anbieter gesucht, die in eine der Kategorien passten. Dank der Informationsplatform Reuters [7] wurden diese auch schnell gefunden:

> Russian authorities said on Sunday they had charged two men over the killing of Kremlin critic Boris Nemtsov and said one of them was a former senior policeman from the mainly Muslim region of Chechnya who had confessed to involvement in the crime.
>
> As fighting in Ukraine appears to intensify and separatists conduct a referendum on independence, observers in the United States, Europe, Russia and elsewhere are increasingly thinking and talking about a second Cold War between America and Russia. But would such a confrontation truly be a second Cold War or would it in fact be something else entirely? These seven key differences suggest that if there is a prolonged struggle between Washington and Moscow, it may not work in the ways that many seem to expect. Some of the differences would be good for America, some would be bad—and one could be ugly.

Diese zwei Artikel sollten nun zu den bereits bestehenden Artikeln zugeordnet werden. Hier wurde zuerst eine menschliche Zuordnung und dann eine Zuordnung mittels dem entwickeltem Algorithmus getätigt. Wie man unschwer erkennen kann, ist die Zuordnung durch den Algorithmus sehr gut. Wir haben in beiden Beispielen die Wunschresultate erhalten.

Ausschnitt	1	2
Menschliche Zuordnung	6,7,8	9,10
Algorithmische Zurodnung	7	9

4.4.5 Optimierung

Jeder Text besteht aus Wörtern die einen Vektor bilden. Somit hat jeder Text eine Dimension und alle Texte mit deren Vektoren bilden eine Basis. Es war sehr schnell klar, dass die Dimension der Vektoren und dementsprechend der Basis enorm gross werden wird. Da unsere Resultate mit der Cosinus-Ähnlichkeit immer sehr nahe beieinander lagen, kamen gewisse Vermutungen auf. Nach der Recherche wurde bewusst, welchen Einfluss die Dimension auf das Resultat hat. [8].
Eigentlich sollte das Proof of Concept die drei Teile aufzeigen und Frameworks evaluieren. Jedoch wurde klar, dass der 'Fluch der Dimensionalität' eine Erscheinung war, die es zu verstehen galt. Somit wurde ein Text Parser gschrieben um gewisse Phänomene und deren Auswirkungen auf das Resultat zu verstehen. Es sollten Methoden gewählt werden und deren Einfluss auf das Resultat beobachtet werden, welche schnell und einfach zu implementieren sind.
Somit wurde entschieden den Schwerpunkt auf folgende Massnahmen zu legen:

[7] `http:\\www.reuters.com`, 19.04.15
[8] Fluch der Dimensionalität: `http://de.wikipedia.org/wiki/Fluch_der_Dimensionalit\%C3\%A4t`

1. **Satzzeichen entfernen:**
   Alle Satzzeichen sollten entfernt werden, Sonderzeichen wie beispielsweise ein Apostroph in 'it's' sollte ebenfalls gleich bereinigt werden.

2. **Kleinschreibung:**
   Alle Wörter sollen nur kleingeschrieben werden und Grossbuchstaben in Kleinbuchstaben umgewandelt, dies garantiert, dass Wörter am Satzanfang gleich geschrieben werden wie in der Satzmitte.

3. **Modalwörter entfernen:**
   Es ist bekannt, dass Modalwörter wenig Informationsgehalt enthalten. Daher ist es vermutlich eine einfache Massnahme diese zu entfernen.

4. **Partikel entfernen:**
   Partikel haben hohen Wertgehalt im Kontext. Beispielsweise können sie eine Negation ausdrücken. Für eine probabilistische Lösung sind diese aber nur Balast und wurden daher entfernt.

5. **Pronomen entfernen:**
   Pronomen zeigen die Zugehörigkeit an und nützen ohne Kontext sehr weniger. Daher sind sie zu entfernen.

6. **Konjunktionen entfernen:**
   Konjunktionen sind enorm wichtig, um den Textfluss zu erhalten und die Logik in einem Text aufzuzeigen. Jedoch wurde entschieden, auf eine Kontextverarbeitung zu verzichten, daher hat diese Wortklasse keinen Stellenwert.

7. **Präpositionen entfernen:**
   Präpositionen sind wichtig um den Kontext zu erklären, in der gewählten Lösung daher nicht nützlich und werden entfernt.

8. **Plural in Singular umwandeln:**
   Diese Idee wurde verfolgt, als festgestellt wurde, dass gewisse Texte besser angezeigt wurden, bei der Suche mithilfe eines Pluralwortes. Die Implementierung ist nicht ganz Trivial und ist daher noch die letzte Massnahme, die im Proof of Concept angestrebt wurde.

Bereits zu diesem Zeitpunkt existierten weitere Ideen um die Dimension zu reduzieren. Jedoch wurde entschieden, dass es sich um ein Proof of Concept handelt und lediglich der Einfluss beobachtet werden sollte. Wichtig an diesem Punkt war auch zu sehen, wieviel Vorteile die einzelnen Massnahmen bringen würden.

	Massnahme								
	roh	1	2	3	4	5	6	7	8
1	219	197	195	187	170	164	157	144	144
2	44	36	36	35	32	32	31	27	27
3	309	267	251	234	207	194	187	172	172
4	388	340	321	300	275	262	245	229	229
5	328	293	284	269	250	241	229	212	211
6	380	354	344	331	317	303	289	274	274
7	199	182	171	164	152	148	141	130	130
8	725	634	609	590	555	538	522	503	502
9	1043	932	902	878	839	820	805	781	779
10	602	538	521	498	472	458	446	425	424
11	391	359	349	332	310	303	292	276	276
12	251	235	227	212	191	185	174	159	158
13	301	269	258	243	220	212	202	189	188
14	693	603	577	552	522	505	485	457	456
15	557	503	484	465	436	421	405	384	380
Basis	4036	3200	3034	2992	2935	2905	2877	2834	2810
Δ	-	0.20	0.05	0.01	0.02	0.01	0.01	0.01	0.01

Wie unschwer zu erkennen ist, ist die Dimension insgesamt um über 1200 geschrumpft. Erstaunlich ist aber die Feststellung, dass eigentlich alle Massnahmen sehr wenig gebracht haben. Die wichtigsten Massnahmen waren (1) Satzzeichen und (2) Kleinschreibung. Die restlichen Massnahmen werden sich hoffentlich dafür in der Qualität wiederspiegeln.

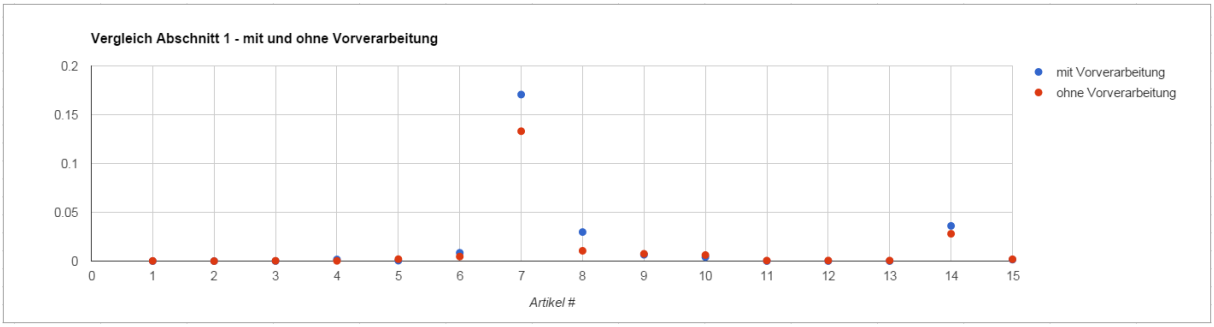


Abbildung 4.2: Diagramm vom ersten Textausschnitt

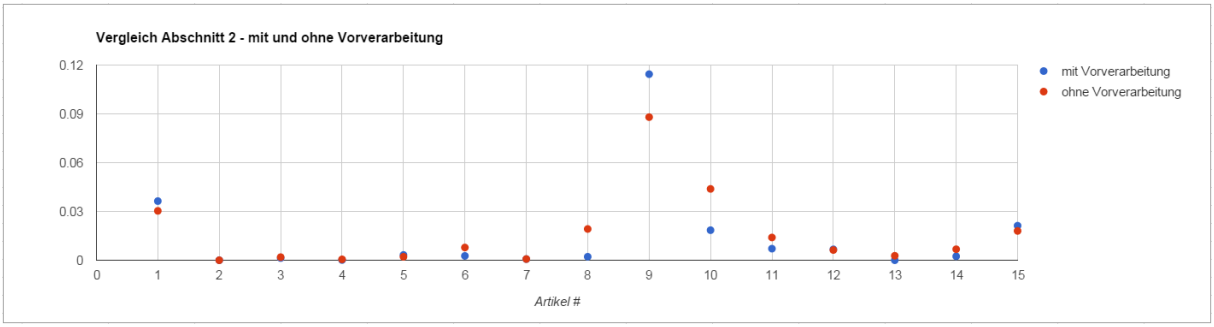


Abbildung 4.3: Diagramm vom zweiten Textausschnitt

Diese zwei Charts visualisieren wie sich die Bereinigung auf den Algorithmus auswirkt. Als Input wurden die 2 gegebenen Textpassagen verwendet und gegen die bestehenden 15 Texte aus der Benchmark-Tabelle gerechnet. Die eingefärbten Punkte entsprechen dem Skalarprodukt ohne das Ausführen der cos-Funktion, welche in der bestehnden Implementierung rechenintensiv ist. Das Ergebnis in blau eingefärbt entspricht den Texten, die Vorverarbeitet wurden. Sämtliche Optimierungsmassnahmen wurden an ihnen ausgeführt. Die roten Punkte entsprechen den Rohdaten vor der Optimierung. Wie man leicht schlussfolgern kann, haben die Massnahmen einen deutlichen Einfluss auf die Qualität der Ergebnisse . Die Punkte separieren sich nach dem Filtern deutlicher und es ist somit ersichtlicher welche Texte zueinander passen.

4.5 Supplier

4.5.1 Einführung

Um die Resultate der Textanalyse dem Benutzer zugänglich zu machen wird eine kleine Webapplikation entwickelt. Sowohl der Cralwer, als auch die Webkomponente sollten in Haskell geschrieben werden. Dafür wurde entschieden, die folgenden Haskell Frameworks zu evaluieren:

- Yesod
- Scotty
- Happstack
- Snap

4.5.2 Vergleichsmethode

Um die Frameworks zu vergleichen, wurden jeweils ein 'Hello World'-Beispiel ausprogrammiert und die Funktionalität des Frameworks analysiert. Da wenig Haskell Erfahrung vorhanden ist, wurde viel Wert auf die Empfehlung aus der Community gelegt. Die Frameworks werden in den folgenden Punkten bewertet:

- **Dokumentation:**
  Wie gut ist die Dokumentation des Frameworks?
- **Community:**
  Wie gut ist die Community? Bietet es einen Forum oder einen IRC Chat um Fragen zu stellen?
- **Nachvollziehbarkeit:**
  Wie nachvollziehbar ist der Code? Wie verständlich ist der Code?

Die oben genannten Punkte werden auf einer Skala von 1 (sehr schlecht) bis 5 (sehr gut) bewertet.

4.5.3 Yesod

Yesod ist das umfangreichste der vier Frameworks. Yesod hat sich zum Ziel gemacht HTML typensicher zu schreiben. Dies hat zur Folge, dass Fehler (z.B. Broken-Link) bereits beim Kompilieren entdeckt werden. Es ist somit möglich Fehler bereits sehr früh abzufangen und nicht erst durch das Ausprobieren im Browser
Ein Beispiel 'Hello World' Programm sieht folgendermassen aus [Sno12]:

```haskell
{-# LANGUAGE OverloadedStrings     #-}
{-# LANGUAGE QuasiQuotes           #-}
{-# LANGUAGE TemplateHaskell       #-}
{-# LANGUAGE TypeFamilies          #-}
import Yesod

data HelloWorld = HelloWorld

mkYesod "HelloWorld" [parseRoutes|
/ HomeR GET
|]

instance Yesod HelloWorld

getHomeR :: Handler Html
getHomeR = defaultLayout [whamlet|Hello World!|]

main :: IO ()
main = warp 3000 HelloWorld
```

Das besondere an Yesod ist, dass relativ viel zur Kompilierzeit generiert wird. Dies ist für den Einsteiger relativ unverständlich, weil sich viel hinter den Kulissen versteckt. Es vereinfacht die Entwicklung, jedoch erschwert es die Fehlersuche enorm. Im obigen Beispiel, werden die Routen mittels 'QuasiQutations' erzeugt, diese zeigen dann auf die HelloWorld Instanz.

- **Dokumenation: 5**
  Zum Yesod Framework wurde ein Buch geschrieben, welches auch frei im Internet verfügbar ist[Sno12]. Darin sind alle Themen, von der Datenbankanbindung bis zur asynchronen Kommunikation, beschrieben.

- **Community: 4**
  Yesod hat eine breite Community, welche Fragen auf vielen Kanälen, wie etwa Google Group oder IRC beantwortet. Weiter sind zur Zeit auf Github 385 Repositories, welche Yesod verwenden oder erweitern, vorhanden. Diese sind sehr nützlich um sich bei der Implementierung auf Beispiele stüzen zu können.

- **Nachvollziehbarkeit: 2**
  Wie schon erwähnt, generiert Yesod relativ viel Code im Hintergrund. Dies ist für den Anfang hinderlich:

  > The mkYesod TH function generates quite a bit of code here: a route data type, parser/render functions, a dispatch function, and some helper types. [Sno12]

4.5.4 Happstack

Happstack ist ein leichtgewichtiges Web Framework, welches auf Geschwindigkeit und Einfachheit fokussiert ist. Durch die schlanke Erscheinung von Happstack unterstützt es nur die nötigsten Funktionalitäten wie Routing, Konfiguration oder Error Handling. Diese Funktionen würden aber unsere Anforderungen eigentlich abdecken. Mit Hilfe einer Template Library kann zum Beispiel HTML sehr einfach geschrieben werden. Dies ist auch der Grund, weshalb die 'Hello World'-Applikation ohne weitere Bibliotheken so einfach aussieht.

```
module Main where

import Happstack.Server (nullConf, simpleHTTP, toResponse, ok)

main :: IO ()
main = simpleHTTP nullConf $ ok "Hello, World!"
```

Ein sehr positives Merkmal ist die einfache Nachvollziehbarkeit mittels Hackage. Die Funktionalität wird sehr einfach und transparent zusammengesteckt ohne Code-Generierung. Dies ermöglicht einen einfachen Einstieg.

- **Dokumentation: 4**
  Für den Anfänger bietet Happstack einen Leitfaden [hap15], welcher ziemlich ausführlich und schrittweise die Verwendung beschreibt. Die Bibliothek selbst ist ziemlich gut auf der Hackage-Seite beschrieben. Es sind zahlreiche Codebeispiele zu den jeweiligen Methoden zu finden.

- **Community: 4**
  Happstack bietet Support auf Stackoverflow und über eine Mailingliste an. Daneben sind zur Zeit 70 Repositories auf Github [9] zu finden.

- **Nachvollziehbarkeit 5:**
  Da Happstack auf Einfachheit setzt sind die Funktionen einfach gehalten und der Code ist sehr verständlich. Zum Beispiel wird im oben angeführten Beispiel ein 'Hello World' zurückggeben, welches als String ohne HTML ausgegeben wird. Für die HTML Ausgaben, muss eine Bibliothek verwendet werden, welche diesen durch einen Funktionsaufruf ausgibt.

4.5.5 Snap

Snap ist vom Umfang her vergleichbar mit Yesod. Ähnlich wie bei Yesod werden Snap Programme mittels dem 'snap init'-Befehl erzeugt. Dieser erzeugt die nötige Dateistruktur. Snap hat ein klar definiertes API, welches auf der Serverseite verwendet werden kann. Dieses API kann auch mit dem Java Servlet API verglichen werden.
Ein einfaches 'Hello World'-Programm sieht folgendermassen aus:

[9] Stand 28.03.15

```haskell
import Snap

site :: Snap ()
site = writeText "Hello Pragmatic Haskeller!"

main :: IO ()
main = quickHttpServe site
```

Dies sieht auf den ersten Blick sehr verlockend aus, jedoch erfordern komplexe Beispiele mehr Einarbeitungszeit. Wie bei Yesod ist die Komplexität doch etwas höher und könnte für den Einsteiger ein Risiko darstellen.

- **Documenation 4:**
  Snap bietet eine ausführliche Dokumentation[sna15] an. Neben Themen wie Datenbankanbindung oder Routing werden auch Anleitungen zum HTML-Templating gezeigt.
- **Community 4:**
  Da Snap eines der ältesten Frameworks ist, ist die Community sehr gross. Dies sieht man schon den 244 Repositiories[10] die auf Github vertretten sind.
- **Nachvollziehbarkeit 3:**
  Der Code ist ähnlich schlank wie Happstack. Jedoch, werden auch bei fortgeschrittene Themen auch auf starke Abstraktion und Typisierung gesetzt.

4.5.6 Scotty

Scotty ist ein minimales Framework und verkauft sich mit den folgenden Features[11]:

- Scotty is the cheap and cheerful way to write RESTful, declarative web applications.
- A page is as simple as defining the verb, url pattern, and text content.
- It is template-language agnostic. Anything that returns a text value will do.

Die minimalistische Herangehensweise des Scotty Framework ist auch in einem 'Hello World' Beispiel ersichtlich:

[10] Stand 28.03.15
[11] `https://hackage.haskell.org/package/scotty-0.4.3`, 20.05.15

```haskell
{-# LANGUAGE OverloadedStrings #-}
import Web.Scotty

import Data.Monoid (mconcat)

main = scotty 3000 $ do
    get "/" $ do
              html $ mconcat ["Hello World"]
```

Daher viel die Bewertung wie folgt aus:

- **Dokumentation 2:**
  Wenig vorhanden. Es existieren nur wenige Tutorials, und keine richtige Dokumentation [Bha15] .
- **Community 2:**
  Von allen bewerteten Frameworks, hat Scotty mit 71 Repositories auf Github die niedrigste Verbreitung.
- **Nachvollziehbarkeit 5:**
  Scotty setzt auf einfache Haskell-Konzepte und ist dadurch sehr verständlich. Die wenigen verfügbaren Funktionen sind gut dokumentiert.

Auch wenn Scotty für den Anfang sehr einfach ist, existiert keine Dokumentation über die Datenbankanbindung. Dies entspricht auch dem Credo von Scottty schlank und einfach zu bleiben. Wie bie Happstack müssen hier entsprechende Bibliotheken hinzugezogen werden, um zum Beispiel sauberen HTML-Code zu schreiben.

4.5.7 Entscheidung

Zusammenfassend kann man sagen, dass folgende Bewertung zu stande kommt:

	Dokumentation	Community	Nachvollziehbarkeit	Summe
Yesod	5	5	2	**12**
Scotty	2	2	5	**9**
Happstack	4	4	5	**13**
Snap	4	4	3	**11**

Da die Bewertung doch sehr knapp zwischen 3 Frameworks ausfällt, ist es nötig noch weitere Aspekte hinzuzuziehen.

- Von unterschiedlichen Haskell Entwicklern wurde **Yesod** als komplex bezeichnet. Es ist somit für Einsteiger nicht sehr geeignet um erste Schritte in Haskell und der Webentwicklung zu machen

- **Scotty** ist sehr einfach und wäre für den Anfang geeignet. Da aber bereits konkrete Vorstellungen existieren, ist anzunehmen, dass gewisse Anforderungen die Möglichkeiten des Frameworks überschreiten.
- **Snap** ist ebenfals einfach und gut dokumentiert. Leider ist es sehr stark dem Servlet API nachempfunden und bietet wenig neue "Konezpte".
- Die Einstiegsschwierigkeiten bei **Happstack** sind gering und somit für Unerfahrene geeignet. Weiter bietet das Framework gute Möglichkeiten für eine Erweiterung und würde somit die Vorstellungen decken.

Aus oben genannten Gründen wurde **Happstack** gewählt.

4.6 Fazit

Das Proof Of Concept war sehr lehrreich. In den folgenden Abschnitten werden die wichtigsten Punkte festgehalten.

4.6.1 Haskell

- Cabal stellt sich als mühsam heraus, wenn mehrere Dependencies angezogen wurde. Manche transitive Abhängigkeiten wurde nicht kompiliert, welche zur Folge hatte, dass der Fehler in einem fremden package gesucht und korrigiert werden musste.
- Fehlersuchen ohne Stacktraces ist unmöglich. Mit Raten der Ursache und einschränke auf einen Code-Bereich ist es einfacher - aber auf langer Sicht nicht praktikabel.
- GHCi eignet sich sehr gut für den Anfänger um Bibliotheken auszuprobieren.
- Viele Haskell-Beispiele im Internet sind nur schwer nachvollziehbar, da diese zu wenig kommentiert oder einfach zu abstrakt sind.
- Als Haskell-Anfänger sind manche Bibliotheken nur mit Beispielen verwenderbar. Viele setzten auf Arrows und Monaden, was den Zugang erschwert, da zuerst diese Konzept eingeübt werden müssen.
- Haskell-Compiler ist streng. Da bei jeder kleinen Änderung meist ein Typenfehler entsteht. Dies ist frustrierend beim Entwickeln.
- Jetbrains Intellij ist als Umgebung nicht hinreichend. Andere IDEs existieren zwar sind diese jedoch nicht wirklich ausgereift.
- Es besteht noch Bedarf an Werkzeugen die den Arbeitsprozess unterstüzen. Der Workflow über das Terminal ist zeitraubend.

4.6.2 Crawler

- Es hat sich gezeigt, dass sich mit Haskell relativ schnell ein Webservice anbinden lässt.
- Die 'lazy evaluation' von Haskell ist nicht fördernd bei der Fehlersuche. Anfänglich wurden die Artikel nicht in die Datenbank geschrieben, später stellte sich als Ursache heraus, dass durch die 'lazyness' nichts ausgeführt wurde.
- PostgreSQL Anbindung mit Haskell ist gut dokumentiert und lief ohne Probleme.

4.6.3 Wikipedia Datenimport

- XML-Parsen mit Tagsoup ist einfach, wenn man verstanden hat, wie Tagsoup funktioniert. Tagsoup sieht die XML-Struktur als eine kontinuerliche Liste von Tags an. Zum Beispiel resultiert der Tag '<title>myarticle.io</title>' in drei Einträge (Start-Tag, Inhalt-Tag, End-Tag).

- Die Wikipedia XML-Datenstruktur ist konsistent und fehlerfrei. Die Wikipedia-Dumps und deren Inhalte sind gut dokumentiert.

4.6.4 Evaluator

- Die Method zum Vergleich von Artikle liefert zuverlässige Resultate.
- Die Dimensionen der Artikel konnten ein Problem werden. Auch wenn versucht wurde diese mit zu verkleiren, sollte bedacht werden, das die Benchmarks lediglich mit 15 Artikeln durchgeführt wurden.
- Die Berechnung der Vergleichgsmatrix ist nicht wie vermutet langsam, auch wenn die Länge bei 2810x15 (Wörter x Artikel) liegt.

4.6.5 Supplier

- Happstack hat sich als gute Wahl herausgestellt, da deren Dokumentation für das Projekt ist. Es existieren jedoch wenige Beispiele im Internet mit Happstack
- Happstack oder Haskell scheint die Resultate zu cache, denn nachdem der Datensatz in der DB abgeändert wurde, werden beim neu Laden mit Browser stets die alten Werte angezeigt. Dies ist für das Projekt kein Problem, dennoch interessant im Hinterkopf zu behalten.
- Happstack ist für ein Webframework beschränkt an Funktionalitäten, es müssen weitere unabhängige Libraries dazugezogen werden, um etwa das Routing zu machen.
- Codebeispiel bei Manchen stellen setzten of Monoide, diese sind nicht verständlich und stammen aus dem Tutorial.

5 Produktifizierung

5.1 Einführung

Ein Ziel der Arbeit ist es ein Produkt zu entwickeln, mit welchem der Benutzer Hintegrund-Informationen und Kontext zu einem Artikeln erhalten kann. Mit dem Proof of Concept wurde gezeigt, dass der Artikelvergleich, wie geplant umsetzbar ist. Ebenfals wurde gezeigt, dass sich Wikipedia als Datenquelle einfach einbinden lässt. Der Grundstein für das Produkt war gelegt, jedoch muss noch bestimmt werden, welche Funktionen nötig sind, und was innerhalb des gesetzten Zeitrahmens machbar ist. Da diese Arbeit explorativer Natur ist, können aus Zeitgründen nicht alle Funktionen umgesetzt werden. Es wird dennoch versucht eine Liste zu erstellen, welche zwingend notwendig zu implementieren sind.

5.2 Welche Informationen sind nötig?

Wenn man einen Artikel im Internet liest, stellt sich meist nicht die Frage in welchem zeitlichen Zusammenhang dieser Artikel steht. Meist wird auch nicht beachtet, wie alt der Artikel ist, erst wenn einem das Thema weiter interessiert, sieht man genauer hin. Beim Lesen des Artikels stoppt man meist, um die fehlenden Informationen zu googln.
Beim lesen des folgenden 'The Guardian' Abschnittes wurde schnell klar, welche Informationen von Nutzen sein könnten.

> "Minutes after Mr Nemtsov's death, the Kremlin's propaganda machine was already hard at work. Mr Putin himself immediately called the killing an act of "provocation" destined to harm his reputation. After Ms Politkovskaya's death, he had commented that she was too unimportant to warrant an assassination, therefore no blame could be laid on him or his power structures. It is easy to understand why Mr Nemtsov had indicated in an interview, 10 days before his death, that he was worried Mr Putin might have him killed. A longtime activist for democracy since his participation in the early 1990s Boris Yeltsin government, Mr Nemtsov was part of Russia's small but active liberal establishment which had supported the 2011-12 mass street demonstrations against Mr Putin's return to the Kremlin. Mr Nemtsov was ready to replace the leader of this movement, the imprisoned blogger and anti-corruption activist Alexei Navalny, on another demonstration in Moscow. He also intended to come out with a detailed denunciation of Mr Putin's involvement in the breakout of war in Ukraine – a role the Kremlin has denied all along."[1]

[1] `http://www.theguardian.com/commentisfree/2015/mar/01/guardian-view-boris-nemtsov-watershed-russia`, 25.05.15

Es stellen sich viele Fragen, wenn man diesen Abschnitt liest:

- Wer ist Mr. Nemtsov, Boris Yeltsin oder Politkovskaya?
- Was haben Putin und Nemtsov für eine Vorgeschichte?
- Welche anderen Berichte gibt es bezüglich der Demonstration?
- Wo liegt die Ukraine? Wie ist das Verhältnis zu Moskau?
- Wie alt ist dieser Artikel? Gibt es neuere Informationen über das Attentat?
- Wann wurde der erste Artikel über den Tod von Boris Nemtsov geschrieben?

Mit etwas Zeit und Recherche kann man jede dieser Fragen beantworten. Jedoch wäre es sehr hilfreich, wenn das Produkt diese Informationen direkt bietet und den Leser unterstützt.

5.3 Funktionen

5.3.1 Brainstorming

Von Anfang an wurde auf eine Web-Applikation gesetzt. Es wurde überlegt, welche Funtkionen den Leser am meisten Unterstüzen könnten und diese wurden in zwei Kategorien aufgeteilt.

Hauptfunktionen

Aus den oben erarbeiteten Ideen wurden folgende als 'must-have' für ein Produkt definiert. Diese umfassen auch den Umfang dieser Arbeit:

- **Suchen mit URL**
  Der Benuzter sollte in der Lage sein durch eine URL eines beliebigen Artikel zu ähnlichen Artikeln zu gelangen.
- **Suchen mit Stichworten**
  Es soll aber auch möglich sein, mit dem Artikelinhalt selbst zu suchen.
- **Ähnliche Artikel chronologisch Auflisten**
  Dies ist die Kernfunktion der Applikation. Artikel die ähnlich sind sollten chronologisch aufgelistet werden um die Entwicklung eines Sachverhalts nachvollziehen zu können.
- **Wikipedia Background Link**
  Es sollen Wikipedia Artikel angezeigt werden, welche mit dem gesuchten Artikel in Zusammenhnag stehen.

Als 'non functional requirement' (NFR) wurde lediglich die Anforderung gestellt, dass die Suche nicht länger als 3 Sekunden dauern darf.

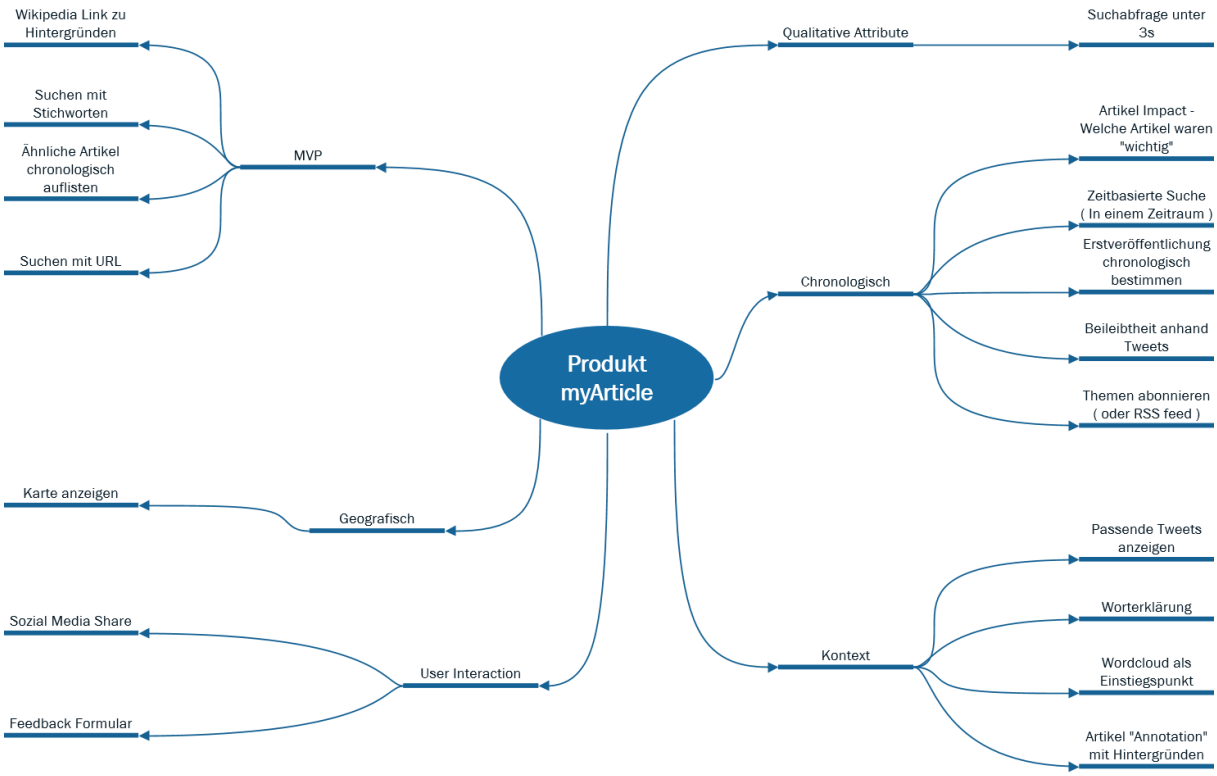


Abbildung 5.1: Funktionen

Optionale Funktionen

- **Karte anzeigen**
  Bei manchen Artikeln ist es möglich einen Karte anzuzeigen, zum Beispiel, wenn ein Land erwähnt wird.
- **Sozial Media**
  Es soll möglich sein Artikel und deren Kontext auf Social Media Plattformen wie Facebook oder Twitter zu teilen.
- **Feedback Formular**
  Der Benutzer soll die Möglichkeit haben Verbesserungsvorschläge zum Produkt anzubringen.
- **Artikel Impact**
  Es soll ersichtlich sein welchen 'Einfluss' die Artikel haben. Beispielsweise wie oft diese auf Twitter geteilt wurden oder welche anderen Quellen diesen Link referenzieren.
- **Zeitbasierte Suche**
  Der Benutzer soll die Möglichkeit haben, den Zeitraum einzuschränken, zum Beispiel alle Artikel aus dem Jahr 2008 zum Thema Terror.
- **Erstveröffentlichung chronologisch bestimmen**
  Wenn zum Beispiel mehrere Artikel zum Theme Boston Bombing angebeben wurden, kann man bestimmen, welcher Artikel die Erstberichterstattung gemacht hat.
- **Beliebtheit anhand Tweets**
  Twitter kann als Datenquelle verwendet werden um die Beliebtheit eines Artikels zu bestimmen.
- **Auf Thema abonnieren**
  Artikel mit einem Ähnlichen Thema sollen als RSS-Feed angeboten werden.
- **Passende Tweets anzeigen**
  Zu Artikeln können die passenden Tweets abgerufen werden.
- **Worterklärung**
  Bei gewissen Wörtern können Erklärungen oder Hintergrundinformationen angezeigt werden (Beispiel: Hagia Sophia)
- **Wordcloud als Einstiegspunkt**
  Anhand einer Wordcloud können brandaktuelle Themen direkt als Suchkriterium angeklickt werden.
- **Artikel 'Annotieren' mit Background Informationen**
  Artikel oder gewisse Passagen können durch Annotationen ergänzt werden.

5.3.2 Mockup

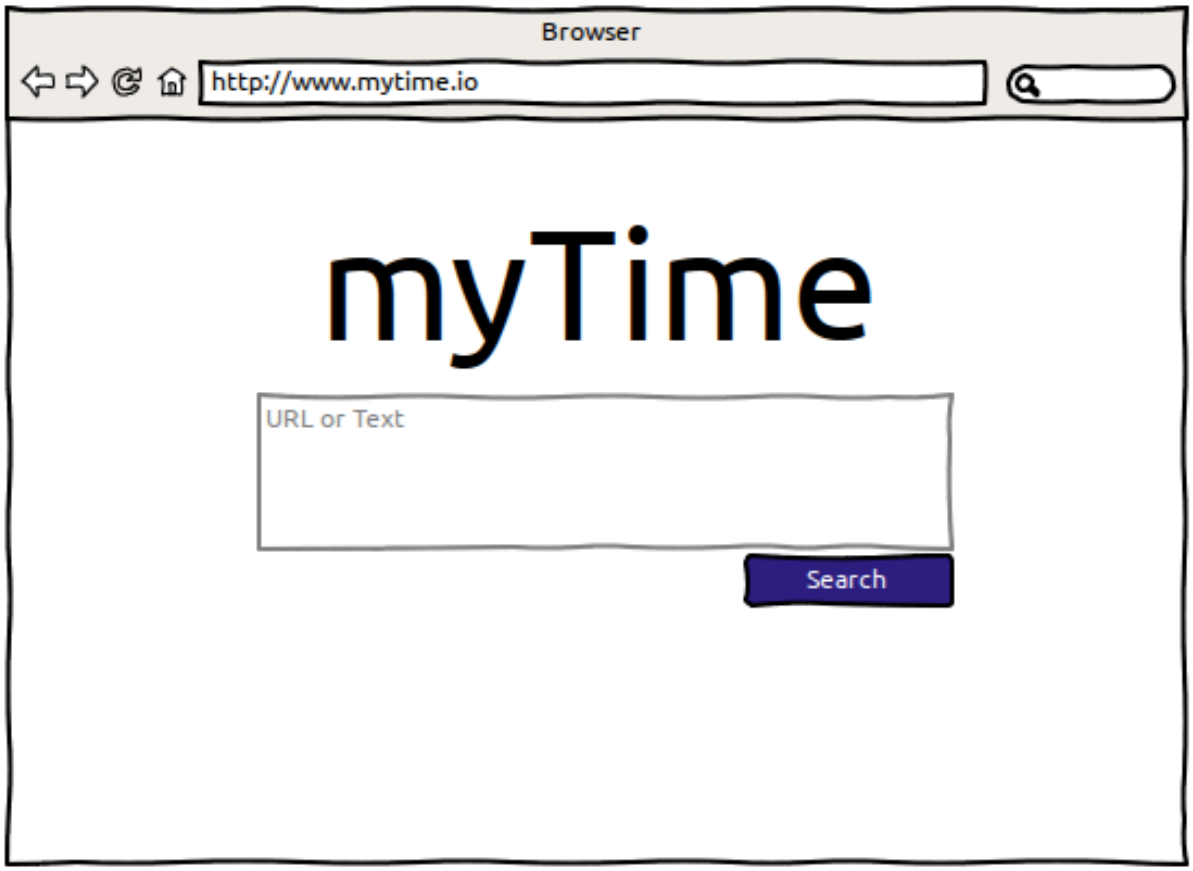


Abbildung 5.2: Mockup der Startseite

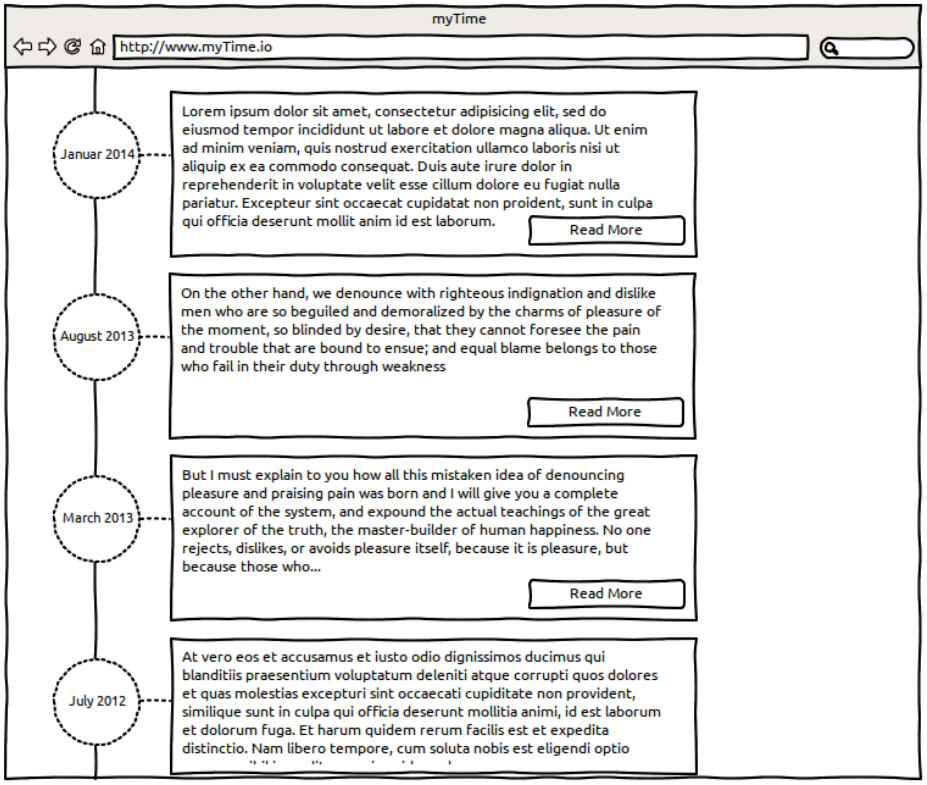


Abbildung 5.3: Mockup der Resultatseite

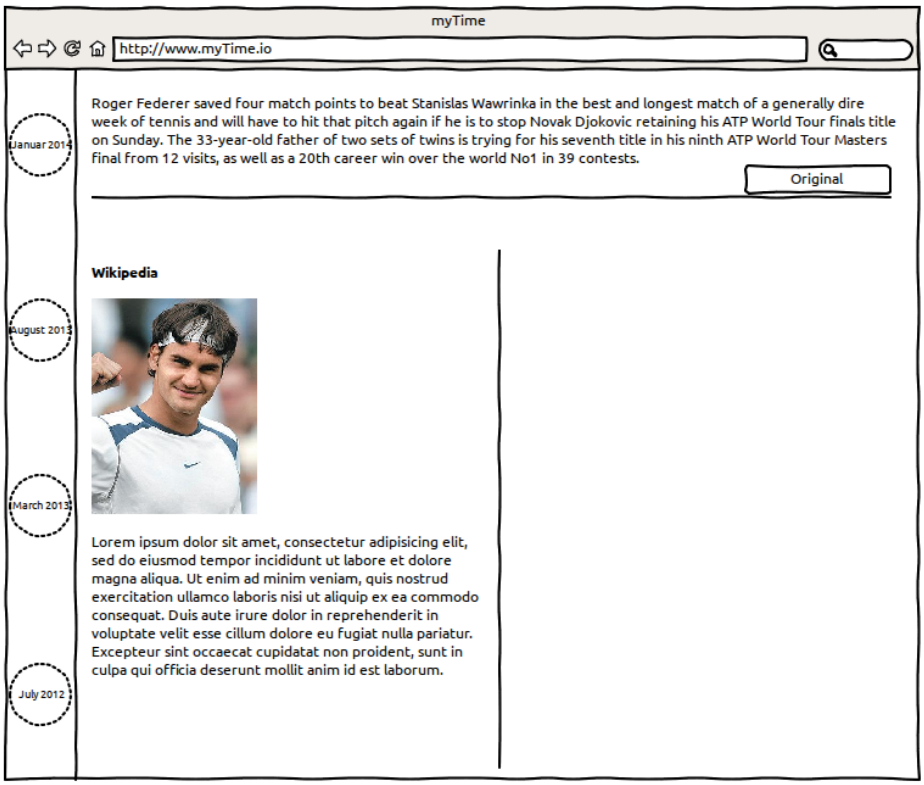


Abbildung 5.4: Mockup eines spezifischen Suchresultats

6 Implementierung

6.1 Einleitung

Nachdem nun der Umfang des Produktes festgelegt wurde, konnte mit der Implementierung begonnen werden. Die Implementierung kam Anfangs schlepppend voran, denn mit Haskell mussten auch zahlreiche Herausforderungen überwunden werden.

6.2 Crawler

Der Crawler aus dem Proof of Concept war sehr zuverlässig und funktionierte mit 'The Guardian' als Informationsquelle einwandfrei. Der Vollständigkeit halber ist an dieser Stelle zu erwähnen, dass sich noch Komplikationen mit dem Paging eingeschlichen haben. Es war nicht so einfach lange Artikel zu verarbeiten wie das nach dem Proof of Concept geplant war.
Weiter ist es trotz zahlreicher Bestrebungen mittels readability-Service nicht gelungen 'The New York Times' als Informationsquelle aufzubereiten. Trotz ähnlichem API wie 'The Guardian' war die Anbindung nicht möglich. Die Zeitung erforderte eine gekaufte Ausgabe und passenden Login. Das API sprach sehr gut auf die Zusammenfassung der Artikel an, jedoch waren die Informationen trotz unterschiedlicher Vorgehensweisen gesperrt. Es wurde beschlossen, die Prioritäten auf andere Teile der Implementierung zu legen, da dieser Umstand nur einen geringen Einfluss auf das Produkt hat.

6.3 Evaluator

Der Evaluator wurde im Proof Of Concept aus zeitgründe in Java geschrieben. Ziel war es diesen in Haskell nachzubauen. Während der Entwicklung wurde die Erfahrung gemacht, dass der Wordcounter mit Performance Problemen zu kämpfen hatte. Im nachhinein ist es offensichtlich, denn es müssen alle Artikel aus der Datenbank geholt und verarbeitet werden. Es kam die Idee, dass die Berechnung des Scores für den Vergleich direkt in SQL gemacht werden könnte. Es wurden verschieden Variante ausprobiert mit Abspeichern von Zwischenresultaten in separaten Tabellen und die Performance zu vergleichen. Schlussendlich wurde eine elegante Lösung gefunden.
Bei jedem Artikel wird ein Score berechnet und abgespeichert. Danach wird mit einer weiteren SQL-Abfrage dieser Score 'normalisiert'. Der Vorteil dieser Methode ist, dass der Vergleich der Artikel mit einer SQL-Abfrage durchgeführt werden kann, und auch gleich die besten x-Artikel selektiert werden können.

6.4 Datenbankimport der Wikipedia Daten

Im Proof Of Concept wurde ein einfaches Haskell Programm geschrieben, um die Wikipedia Daten in die Datenbank zu laden. Jedoch wurde es nur mit einem 200MB Dateifragment getestet. Die Annahme, dass die Software auf die 50GB grosse Datei einfach skaliert werden kann, musste schnell verworfen werden.

6.4.1 Unsauber Wikimedia Daten

Nach dem ersten Start mit der 50GB grossen Datei vergingen nur wenige Minuten bis das Haskell Programm folgende kryptische Fehlermeldung ausgab:

```
"Immanuel Kant"
*** Exception:
Error:
"source" (line 2, column 1):
unexpected end of input
expecting lf new-line, "<", "’’’’", "’’’", "[[", "[", ":", "\\[", "\\(",
↪  "&", "{{{", "{{", "{|", "=", "----", "#", "*", ";", ":<math>", white
↪  space or " "
```

Leider lieferte GHC keine weiteren Hinweise. Ein Stacktrace, wie man ihn aus der Java Welt kennt, wäre hier nützlich gewesen. Anhand der Ausgabe war wenigstens zu erkennen, dass der Fehler im Artikel ’Immanuel Kant’ passiert ist. Der Artikel konnte nicht in die Datenbank geschrieben werden.
Der Prozess, um die Wikipedia Daten zu laden ist relativ einfach. Die XML Datei wird ’lazy’ gelesen und die einzelnen Tags werden extrahiert. Aus einer Liste von XML-Tags werden die fürs relevanten herausgefiltert. Dabei handelt es sich hauptäschlich um den Titel und den Text. Der Text wird im WikiMedia Format mitgeliefert, diesert wird mithilfe der Pandoc-Library in einen lesbaren Text konvertiert und in die Datenbank gespeichert.
Um den Fehler zu reproduzieren, wurde eine Datei mit jenem Artikel präpariert. Mit dem GHCi Debugger, wurde nun versucht herauszufinden, welche Methode genau für diesen Fehler verantwortlich war.

```
*WikiDumper> :set -fbreak-on-exception
*WikiDumper> :set -fbreak-on-exception
*WikiDumper> :trace main
"Parse file"
Stopped at <exception thrown>
_exception :: e = _
[<exception thrown>] *WikiDumper> :history
-1  : openingTag:noAttribute (src/WikiDumper.hs:44:25-26)
-2  : openingTag (src/WikiDumper.hs:43:28-38)
-3  : openingTag (src/WikiDumper.hs:43:19-51)
-4  : openingTag (src/WikiDumper.hs:(43,1)-(44,26))
-5  : pageToArticle:takeContent (src/WikiDumper.hs:68:84-101)
-6  : openingTag:noAttribute (src/WikiDumper.hs:44:25-26)
-7  : openingTag (src/WikiDumper.hs:43:28-38)
-8  : openingTag (src/WikiDumper.hs:43:19-51)
-9  : openingTag (src/WikiDumper.hs:(43,1)-(44,26))
-10 : pageToArticle:firstRevision (src/WikiDumper.hs:60:56-76)
-11 : pageToArticle:firstRevision (src/WikiDumper.hs:60:40-85)
-12 : pageToArticle:firstRevision (src/WikiDumper.hs:60:33-85)
-13 : pageToArticle:firstRevision (src/WikiDumper.hs:60:9-85)
-14 : pageToArticle:text (src/WikiDumper.hs:57:42-59)
-15 : pageToArticle:takeContent (src/WikiDumper.hs:68:69-111)
```

Es scheint, dass die Funktion 'pageToArticle' der Verursacher war, denn die 'openingTag'-Funktion ist nur eine Hilfsfunktion, welche eine Datenstruktur erzeugt. Aus dem Stacktrace ist die genaue Ursache nicht klar erkennbar, deshalb wurde der Code manipuliert und die 'wikiToPlainText'-Funktion entfernt. Nach dieser Massnahme hat das Programm die präparierte Datei verarbeitet. Der folgende Code-Abschnitt zeigt die Implementierung der erwähnten Methode:

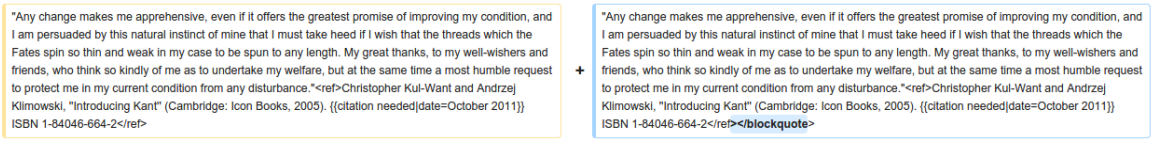
```
wikiToPlainText :: T.Text -> NT.Text
wikiToPlainText htmlText = NT.pack $ writePlain def $ readMediaWiki def $
 ↪  text
    where text = T.unpack htmlText

pageToArticle :: Page -> Article
pageToArticle page = Article (NT.pack $ "Wikipedia" ) (wikiToPlainText
 ↪  source) (wikiToPlainText text)
    where
        source = takeContent "title" page
        text = takeContentFromRevision $ firstRevision page
        -- skippped rest ...
```

Es handelt sich dabei um das Parsen des Textes vom Wikimedia-Format in ein lesbares Text-Format. Die Vermutung war, dass Pandoc die Daten nicht richtig parsen konnte. Um den genauen Fehler zu finden, wurde der Artikel Abschnitt für Abschnitt gelöscht und mit dem Wikidumper überprüft. Bald war klar welcher Abschnitt den Fehler verursachte:

```
:<blockquote>
"Any change makes me apprehensive, even if it offers the greatest promise
 of improving my condition, and I am persuaded by this natural instinct of
  mine that I must take heed if I wish that the threads which the Fates
 spin so thin and weak in my case to be spun to any length. My great
 thanks, to my well-wishers and friends, who think so kindly of me as
 to undertake my welfare, but at the same time a most humble request
 to protect me in my current condition from any disturbance.
 "<ref>Christopher Kul-Want and Andrzej Klimowski, ''Introducing Kant''
 (Cambridge: Icon Books, 2005). {{citation needed|date=October 2011}}
 ISBN 1-84046-664-2</ref>
```

Beim weiteren Studieren, stellte sich heraus, dass der '<blockquote>' nicht sauber abgeschlossen wurde. Nach manueller Korrektur des Artikel war auch klar - dieser Artikel ist nicht in einem sauberen Wikimedia-Format! In der Hoffnung, dass dies eine Ausnahmen sein könnte, wurde der Fehler direkt in der XML-Datei und auf der Wikipedia-Seite [1] korrigiert.



Leider trat genau dieser Fehler nocheinmal an einem anderen Ort auf. Nun musste eine andere Lösung gefunden werden, welche die unsauberen Artikel herausfiltert, ohne das Programm zu stoppen.

6.4.2 Pandoc Fehler

Pandoc ist eine verbreitete Library um unterschiedliche Formate untereinander zu konvertieren. Neben Pandoc als Haskell-Bibliothek ist sie auch als Commandline-Tool verbreitet. Was erstaunlich war, ist dass Pandoc bei einem Fehlerfahl die error()-Funktion aufruft und das ganze Programm stoppt. Dies mag beim Commandozeilen-Tool berechtigt sein, aber für ein Library ist dies unbrauchbar. Es scheint, dass jemand anders schon versucht hat alle Wikipedia Daten zu parsen und gescheitert ist. Auf Github wurde ein Issue[2] erstellt.
Das Issue wurde auch korrigiert, jedoch war seit der Korrektur kein neuer Release herausgegeben worden. Um nun die neuste Version zu verwenden, blieb nichts anders überig als Pandoc selbst zu kompilieren.

[1] `http://en.wikipedia.org/w/index.php?title=Immanuel_Kant&type=revision&diff=662142058&oldid=662025499`, 25.05.15
[2] `https://github.com/jgm/pandoc/issues/1848`, 25.05.15

Pandoc erlaubt in der Entwicklungsversion eine Fehlerbehandlung. Statt einem Abbruch wird nun die Fehlermeldung mitgeliefert. Im Fehlerfall kann nun der ganze Artikel übersprungen werden.

```
pandocToPlain :: Either PE.PandocError P.Pandoc -> Maybe NT.Text
pandocToPlain (Left error) = Nothing
pandocToPlain (Right pandoc) = Just $ NT.pack $ P.writePlain P.def pandoc

wikiToPlainText :: T.Text -> Maybe NT.Text
wikiToPlainText htmlText =  pandocToPlain $ P.readMediaWiki P.def $
 ↪  T.unpack htmlText
```

Nachdem nun alle fehlerhaften Artikel herausgefiltert wurden, kam beim erneuten Testen eine andere Fehlermeldung:

```
 Exception: Prelude.head: empty list
```

Mit dem GHCi-Debugger wurde die P.writePlain Funktion als Verursacher ermittelt. Bei genauer Analyse, scheint eine Funtkion in Pandoc nicht sauber mit leeren Listen umgehen zu können.
Dieses Problem war nicht einfach zu lösen, den die entsprechende Funktion zu korrigieren mislang. Es konnte mit dem Fehler aber umgegangen werden, so dass das Problem keine weiteren Auswirkungen auf die Arbeit hatte.

6.5 Supplier

Aus dem Proof Of Concept wurde Happstack als Webframework evaluiert. Happstack bietet einen fertigen Routing Mechanismus an und für das generieren des HTML Codes werden externe Bibliotheken verwendet. Die Funktionsweise ist ähnlich wie die einer Templating-Engine (Bsp. TWIG, Play). Die im Proof of Concept verwendete Bibliothek war HSP (Haskell Server Pages) und wurde aufgrund des guten Tutorials verwendet. Das Happstack Tutorial machte HSP für Anfänger zugänglich.
HSP wird verwendet um aus einem String eine XML-Struktur zu generieren. Anfänglich wurde diese Bibliothek verwendet, um die HTML-Struktur aufzubauen. Das folgende Beispiel zeigt, wie mittels dem 'hsx' QuasiQuoter die interne Struktur generiert wurde.

```
page :: (XMLGenerator m, StringType m ~ Text) => XMLGenT m [ChildType m]
page = [hsx|
        <%>
        <div style="padding-top: 195px"></div>
        <div class="container">
            <div class="col-md-3"></div>
            <div class="col-md-6">
            </div>
            <div class="col-md-3"></div>
        </div>
        </%> |]
```

Der QuasiQuoter generiert einen Monad-Transfomer 'XMLGentT a b', wobei 'b' mit [ChildType m] belegt ist. Da es sich hier um mehrere HTML-Tags handelt, muss aber der Typ a bestimmt werden. Dieser wurde auf die Klassen 'XMLGenerator m' und 'StringType m' begrenzt. Die Frage, die sich nun stellte war, wofür der Ausdruck ' Text' steht. Es handelt sich hierbei um einen Hinweis an den Compiler. Dieser soll darauf achten, dass 'StringType m' auf einen Wert vom Typ Text abgeleitet wird. Dies erscheinte doch etwas fragwürdig und komplizert typisiert.
Ein Problem entsteht nun wenn ein Teil des HTML-Codes in eine seperate Funktion ausgelagert werden soll.

```
anotherChildElement :: (XMLGenerator m, StringType m ~ Text) => XMLGenT m
↪  [ChildType m]
anotherChildElement = asChild ("Hello" :: String)

page :: (XMLGenerator m, StringType m ~ Text) => XMLGenT m [ChildType m]
page = [hsx|
        <%>
        <div style="padding-top: 195px"></div>
        <div class="container">
            <div class="col-md-3"></div>
            <div class="col-md-6">
                <% anotherChildElement %>
            </div>
            <div class="col-md-3"></div>
        </div>
        </%> |]
```

Darauf meldete sich der Compiler mit einer langen Fehlermeldung.

```
src/HomePage.hs:16:23:
    Overlapping instances for EmbedAsChild m String
      arising from a use of ‘asChild’
    Matching instances:
      instance [overlap ok] EmbedAsChild m c => EmbedAsChild m [c]
        -- Defined in ‘HSP.XMLGenerator’
      instance (Functor m, Monad m) =>
               EmbedAsChild (Web.Routes.RouteT.RouteT url m) String
        -- Defined in ‘Web.Routes.XMLGenT’
    (The choice depends on the instantiation of ‘m’
     To pick the first instance above, use IncoherentInstances
     when compiling the other instance declarations)
    In the expression: asChild ("Hello" :: String)
    In an equation for ‘anotherChildElement’:
        anotherChildElement = asChild ("Hello" :: String)
```

Das Problem lag daran, dass der Typ 'm' bei allen Funktionen übereinstimmen musste. Bei diesem einfachen Beispiel ist die Korrektur noch einfach, jedoch wenn mehrere Funktionen verkettet wurden war das Beheben umständlich.
Nachdem dieses Problem einige Male aufgetaucht und erfolgreich gelöst wurde, wurde beschlossen nach Alternativen zu suchen. Die Lösung mit HSP war zwar syntaktisch und stylistisch schön, jedoch war der Gebrauch sehr umständlich. Eine Alternative bietete blaze-html. Diese Bibliothek wurde auch im Happstack Tutorial erwähnt, jedoch aus Gründen der Lesbarkeit zu Beginn aussortiert.
Das folgende Codebeispiel zeigt die HTML Generierung mittels blaze-html:

```
page  :: H.Html
page  = H.div ! A.style "padding-top 195px" $ do
            H.div ! A.class_ "container" $ do
                H.div ! A.class_ "col-md-3" $ mempty
                H.div ! A.class_ "col-md-3" $ do
                    H.img ! A.src "myarticle.png" ! A.class_
                     ↪  "img-responsive" ! A.alt "Responsive image"
                    H.form ! A.method "GET" ! A.action "result" $ do
                        H.div ! A.class_ "input-group" $ do
                            H.input ! A.type_ "text" ! A.class_
                             ↪  "form-control" ! A.placeholder
                             ↪  "Search for .."
                                    ! A.name "searchValue"
                            H.span ! A.class_ "input-group-btn" $ do
                                H.button ! A.class_ "btn btn-default" !
                                 ↪  A.type_ "submit" $ do
                                    "GO!"
```

Zugegebenermassen ist der Code nicht so einfach lesbar wie reines HTML. Jedoch bietet es eine einfachere Typiseriung, denn alle HTML Elemente sind nur vom Typ 'Html' und nicht vom Typ des Inhalts abhängig. Daneben können die Methoden auch einfacher zusammengeschraubt werden. Dies führte dazu, dass bewusst auf blaze-html gesetzt wurde.

6.6 Qualitätsicherung

6.6.1 Unittests

Es wurde entschieden für das Unit-Testing Hspec[3] zu verwenden. Um die Qualität genügend zu sichern hat Prof. Dr. Josef. M. Joller eine Testabdeckung von 30% etwa empfohlen. Es wurde versucht sich an diesen Wert zu halten und nur in begründeten Fällen davon abzuweichen.
Um Unit-Tests in Haskell zu verdeutlichen folgt hier ein kleines Codebeispiel:

[3] `http://hspec.github.io/`, 25.05.15

```haskell
spec :: Spec
spec = do
    describe "DatbaseConnectionTest" $ do
        it "Should be equal" $ do
          let a = Article "20min.ch" "Some text"
          let b = Article "20min.ch" "Some text"
          a `shouldBe` b

        it "should insert new article" $ withDatabaseConnection $ \conn
         ↪  -> do
          truncateArticle conn
          reslt <- insertArticle conn (Article "20min.ch"
           ↪  "Article Text ...")
          count <- countArticles conn
          count `shouldBe` [(Only 1 )]
```

Wie man Anhand des Beispiels sehen kann, sind die Tests relativ einfach lesbar und bereiten Java-Programmierern wenig Mühe.

6.6.2 Coverage

Für die Berechnung der Testabdeckung wurde das HPC (Haskell Programm Coverage)-Tool verwendet. Auch wenn dieses in GHC eingepflegt ist, ist die Dokumentation spärlich. Es mussten mehrere Versuche gestartet werden, da HPC auch die Testabdeckung in den Tests überprüft. Die foglenden Graphiken zeigen die Testabdeckung der einzelnen Module, wobei logischerweise die Test herausgefiltert wurden. Wie aus der Grafik ersichtlich ist, wurde das Ziel von 30%

module	Top Level Definitions		Alternatives		Expressions	
	%	covered / total	%	covered / total	%	covered / total
module ArticleDao	75%	15/20	-	0/0	78%	81/103
module DatabaseAccess	70%	12/17	-	0/0	83%	55/66
module HttpRegeustSenderSpec	50%	1/2	-	0/0	90%	19/21
module HttpRequestSender	50%	1/2	50%	1/2	14%	4/28
module PersonDao	84%	16/19	-	0/0	100%	85/85
Program Coverage Total	75%	45/60	50%	1/2	80%	244/303

Abbildung 6.1: Testabdeckung für 'commons'

erfüllt, jedoch sind einige Punkte anzumerken:

- In der 'web' Komponente wurden für die Webseiten keine Tests geschrieben, da diese einen HTML-Output liefern.
- Die Zahlen müssen mit bedacht genossen werden, denn HPC bestimmt die Testcoverage nur für einen Modul. Wenn vom crawler-Modul auf eine Funktion in commons-Modul verwendet wird, ist die Verwendung nicht ersichtlich. Somit sollte tendenziell die Abdeckung im commons-Modul höher liegen.

module	Top Level Definitions		Alternatives		Expressions	
	%	covered / total	%	covered / total	%	covered / total
module Crawler	0%	0/46	0%	0/12	0%	0/249
module Main	100%	2/2	-	0/0	100%	19/19
module ReadabilityApi	30%	6/20	50%	1/2	58%	41/70
module WikiDumper	76%	16/21	45%	5/11	91%	188/205
module WikipediaPersonHelper	100%	14/14	60%	6/10	97%	167/172
Program Coverage Total	36%	38/103	34%	12/35	58%	415/715

Abbildung 6.2: Testabdeckung für 'crawler'

module	Top Level Definitions		Alternatives		Expressions	
	%	covered / total	%	covered / total	%	covered / total
module WordCounter	85%	17/20	25%	1/4	84%	188/223
Program Coverage Total	85%	17/20	25%	1/4	84%	188/223

Abbildung 6.3: Testabdeckung für 'evaluator' ('wordcounter')

- Der Crawler wurde bewust nicht getestet, da dieser einen externen Service anspricht. Diesen zu 'mocken' ist umständlich und würde keine grossen Vorteile bringen.

module	Top Level Definitions		Alternatives		Expressions	
	%	covered / total	%	covered / total	%	covered / total
module Countries	57%	11/19	100%	3/3	53%	468/874
module Page.AboutPage	0%	0/1	-	0/0	0%	0/67
module Page.HomePage	0%	0/1	-	0/0	0%	0/102
module Page.OneArticlePage	0%	0/12	0%	0/4	0%	0/499
module Page.ResultPage	0%	0/3	-	0/0	0%	0/280
module Page.Template	0%	0/2	-	0/0	0%	0/78
module Routing	0%	0/109	0%	0/12	0%	0/198
Program Coverage Total	7%	11/147	15%	3/19	22%	468/2098

Abbildung 6.4: Testabdeckung für ’web’ (’supplier’)

7 Finalisierung

7.1 Einleitung

7.2 Lessons Learned: Software Engineering

7.2.1 Neue Programmiersprache

Auch wenn man einige Jahre Erfahrung im Programmieren hat darf nicht unterschätzt werden, wie zeitraubend der Einstieg ist. Die Sprache selbst zu erlernen ist das Einfachste. Was einem auch fehlt, ist das Wissen wie man Fehlermeldungen liest oder wie man Probleme löst.
Am Anfang wurde in Haskell viel mittels copy-paste gelöst und angepasst. Bei Fehlern hat sich Frust bemerkbar gemacht und es wurde versucht Fehlermeldungen nachzuvollziehen. Mit der Zeit entwickelte sich eine Vorahnung. Dies führte während der Entwicklung des Produkts zu folgendem Arbeitsprozess. Zuerst wurde versucht eine Lösung für das eigentliche 'Problem' zu finden. Anschliessend wurde dann die Lösung in Haskell umgesetzt. Dies bereitete oft Mühe und es mussten Beispiele aus dem Internet zurate gezogen werden.

7.2.2 Proof Of Concept

Das Proof Of Concept hat sich als gut erwiesen, denn es wurden viele Fehler gefunden und aufgezeigt. Viele Risiken wurden minimiert. Auch wurde schnell klar, wo die Hürden bei Haskell liegen könnten.

7.3 Lessons Learned: Haskell

7.3.1 Fehlende Tools

Wie im Proof Of Concept festgestellt werden mussten, fehlte es an Tools für die professionelle Haskell Entwicklung. Mit 'vim' und Texteditor lassen sich zwar einfache Beispiele schreiben, jedoch für einen grösseren Umfang benötigt es eine gute Entwicklungsumgebung.
Mit IntelliJ und den Plugins HaskForce-Plugin [1] und FileWatchers lässt sich die Entwicklung beschleunigen, jedoch ist dies nicht annähernd so ausgereift wie etwa in Java.
Neben dem IDE fehlt es auch an einem guten Debuggen. Denn mit dem GHCi kann der Code debuggt werden, jedoch ist dies umständlich.

[1] `https://github.com/carymrobbins/intellij-haskforce`, 28.06.15

7.3.2 Nicht lesbarer Code

Das Problem des unsauberen Codes existiert in jeder Sprache. Jedoch gibt es in Haskell sehr viele Möglichkeiten dasselbe Problem zu lösen. Ein einfaches Beispiel ist das importieren von Modulen. Es existieren folgende Möglichkeiten:

```
import qualified Data.Text as T
import           Data.List (isInfixOf)
import           Data.Int
```

Jeder Entwickelt präferiert eine dieser Varianten. Meist wird einfachheitshalber die letzte Variante genommen, bei diesem werden alle Funktionen importiert. Beim Schreiben mag dies angenehm sein, jedoch ist das Nachvollziehen für einen fremden Entwickler schwierig. Dies ist genau das Problem, denn es existieren viele Beispiele im Internet, welche alle Möglichkeiten verweden.

7.3.3 Verwirrung mit ByteString, String, Text, LazyText

Es existieren viele Möglichkeiten String in Haskell zu representieren. Jeder hat seine Vor- und Nachteile. Das Problem ist, dass manche Libraries nur die eine Art von Strings verwenden und es müssen zwischen den Typen konvertiert werden. Der StringOverloaded Extension macht dies einfacher, jedocht ist es immernoch ein Mehraufwand.

7.3.4 Zeitdarstellung ist kompliziert und uneinheitlich

Wie in anderen Programmiersprachen ist auch in Haskell das Darstellen von Zeit nicht sauber gelöst. Es existieren verschiedene Datentypen, um die Zeit zu repräsentieren. Es wurde versucht UTCTime zu verwendet, doch das Problem war nun diesen in die Datenbank zu schreiben. Das Mapping von UTCTime auf den 'TIMESTAMPZ' in Java war noch fehlerhaft.
Deshalb wurde entschieden die Zeit als String zu behandeln und von der Datenbank in ein 'TIMESTAMPZ' zu konvertieren.

7.3.5 Fehlende Interfaces

In Haskell existieren 'Klassen', wie der Name sagt, sind diese primär da um Datentypen, die eine ähnliche Eigenschaft haben in 'Klassen' zusammenzufassen. Zum Beispiel war das Erstellen der Pages in der Web-Komponent immer ähnlich. Es wurden meist Daten aus der Datenbank gelesen und diese später als HTML dargestellt. Es wurde versucht dieses Verhalten in ein 'Interface' zusammenzufassen, um das Routing generisch zu gestallten.
Doch was aus der Java-Welt mittels Strategy-Pattern einfach umzusetzen war, ist in Haskell ungmölich.

7.3.6 Typisierung

Die Typisierung ist sogleich hilfreich und hinderlich. Hilfreich ist die Typisierung, wenn man sie gezielt einsetzt. Diese Aussage trifft aber auch auf andere Sprachen zu. Jedoch hat Haskell eine Spezialität, die auch Seiteneffekte in die Typisierung einfliessen lässt. Dies ist auf den ersten Blick sehr hilfreich. Man kann darauf vertrauen, dass die Methode deterministisch ist oder nicht. Doch wenn der Code eine gewisse Grösse erreicht hat, wird das Refactoring schwieriger. Zum Beispiel wurde beim Crawler ein Bug festgestellt, der dazu führte, dass die Applikation mit einer 'empty-list'-Exception abbricht. Es gab eine Vermutung wo dieser auftreten könnte. Aufgrund der unhandlichen Bedienung des Debuggers hätte man hier an der vermuten Stelle die Parameter mittels 'print'-Aufruf ausgeben können. Was in Java ohne zu überlegen geklappt hätte, hat in Haskell zu einem Compilerfehler geführt. Die Ausführungskette bis zur betroffenen Methode hätte mit einer IO-Monade typisiert werden müssen.

7.3.7 Fehlen der 'Globalen Variable'

Beim WordCounter wurde nachträglich das herausfiltern der Stoppwöter implementiert. Die Stoppwörter kamen dabei aus einer Datei. Die Datei wurde in der Main-Funktion gelesen und an die Funktionen als Parameter weitergegeben, bis zu dem Punkt wo diese gebraucht wurde. In Java wäre der Inhalt der Datei einmalig gelesen und in eine globale Variable geschrieben worden. Dies ist ein typisches Problem in Haskell: In einer tiefen Methode braucht es Zusatzinformationen und diese müssen als Parameter bis zu jener Methode übergeben werden.
Auch wenn manche Parameter als Alias oder Daten-Typen gekapselt werden können, kann dies das Pflegen und Ändern des Codes erschweren.

7.3.8 Fehlende Erfahrung

Ist der geschrieben Code überhaupt funktional. Diese Frage stellte sich während der Entwicklung mehrmals. Es ist bekannt, dass Haskell eine rein funktionale Sprache ist, doch wenn die Möglichkeit existiert mittels do-Block imperativen Code zu schreiben, so tendiert man dazu dies auch zu tun.

7.3.9 Fehlende Beispile

Es fehlen praktische Beispiele im Internet. Zum Beispiel war es schwierig Beispiele zu finden, bei dem eine einfache Webapplikation mittels Datenbank verbunden wurde. Es stellten sich viele Fragen, wie etwa: Wann soll eine Datenbank Verbindung gemacht werden oder wo die Transaktionsgrenze liegt. Fragen die sich einfacher in anderen Sprachen beantworten lassen, sind in Haskell mit vielen Nachforschungen verbunden. Oft findet man Informationen im Internet oder muss sogar als Erster eine Lösung probieren.

7.4 Schlusswort

Das Erlernen von Haskell ist etwas sehr lehrreiches. Es lässt viele Konzepte hinterfragen und auch schärft es das funktionale Denken. Da Haskell einen akademischen Ursprung hat, kommt

man aber schnell mit komplexen Themen in Berührung.
Das Implementieren einer Webapplikation in Haskell ist sicherlich etwas sehr lehrreiches. Es hat sich aber auch gezeigt, dass,Haskell noch nicht für den praktischen Einsatz ausgreift ist. Es ist eine Sprache um Konzepte zu erlernen und es fehlt noch an wichtigen Werkzeugen für einen unterstüzten Entwicklungsalltag.

7.5 Cabal

In der Haskell-Community ist Cabal das einzige Tool um Dependencies zu verwalten. Es ist sozusagen das 'Package Management Tool' von Haskell. Jedoch gibt es auch Stimmen, welche sagen, dass es sich bei Cabal nur um das Build Management handelt. [Mil15]
Warum hier Cabal erwähnt wird, liegt in der Mengen der Probleme, welche bei Cabal üblich sind und die Erkenntnisse die mitgenommen werden konnten.

7.5.1 Versionierung

Wie in jedem Build Management können die Versionen der Libraries angegeben werden. Im Java Umfeld werden die Libraires mit der expliziten Version angegeben. In Cabal hingegen werden die verwendeten Versionen mit einem Constraint angegeben. Wie der folgende Abschnitt aus der Happstack-Bibliothek zeigt, ist dies in der Community der übliche Weg:

```
Build-depends:       base             >= 3.0 && < 5.0,
                     bytestring       == 0.9.*,
                     happstack-server >= 6.0 && < 6.6,
                     harp             >= 0.4 && < 0.5,
                     hsx              >= 0.9.1 && < 0.10,
                     hsp              >= 0.6.1 && < 0.7,
                     mtl              >= 1.1 && < 2.1,
                     utf8-string      == 0.3.*,
                     syb              == 0.3.*,
                     text             >= 0.10 && < 0.12
```

Das Problem liegt nun darin, dass in einem Team verschiedene Entwickler verschiedene Versionen einer Unterbibliothek verwenden. Es ist somit nicht garantiert, dass der Abhänigkeitszustand eingefroren ist. Um die benötigten Versionen zu ermitteln, verwendet Cabal einen Constraint-Solver[2]. Das Problem ist nicht einfach, denn angenommen die 'hsp'-Library im obigen Beispiel setzte auf einen Range von Bibliotheken und Versionen, muss auch diese mit dem Happstack range übereinstimmen. Cabal versucht quasi die beste 'Lösung' an Dependencies zu finden, welche für alle Bibliotheken stimmt.
Das eigentliche Problem ist jedoch, dass Cabal erlaubt einen Versionsbereich anzugeben. Denn was der Entwickler von happstack hier prpogaiert ist, dass die Library mit allen Versionen von 'hsp' zwischen 0.6.1 und 0.7 klarkommt. Zustäzlich wird suggeriert, dass gleichzeitig 'text'

[2] `https://github.com/haskell/cabal/wiki/Constraint-Solving`, 26.05.15

zwischen 0.10 und 0.12 unterstüzt wird. Nun muss für jede Kombination der 9 libraries eine Funktionsgarantie existieren. Die Frage, die sich hier stellt ist wie dies gewährleistet werden kann.

7.5.2 Praktische Lösungen

Cabal hat das Problem erkannt, statt aber den Kern des Problems zu lösen, wurde ein zusätzlicher Mechanismus geschaffen. Versionen können eingeschränkt werden. Mittels dem 'cabal freeze' Befehl können constraints zu expliziten Versionen erstellt werden, dies garantiert auch, dass Entwickler im gleichen Team alle die gliechen Versionen verwenden. Die Abhängigkeiten werden wie im folgenden Beispiel ersichtlich einfach eingefroren:

```
# content cabal.config
constraints: HTTP ==4000.2.19,
             aeson ==0.8.0.2,
             array ==0.5.0.0,
             asn1-encoding ==0.9.0,
             asn1-parse ==0.9.0,
             asn1-types ==0.3.0,
             async ==2.0.2,
             attoparsec ==0.12.1.6,
             base ==4.7.0.2,
             base64-bytestring ==1.0.0.1,
             binary ==0.7.1.0,
             blaze-builder ==0.4.0.1,
             blaze-textual ==0.2.0.9,
             byteable ==0.1.1,
             ...
```

Beim erneuten builden, werden nun die Bibliotheken und deren Versionen aus der 'cabal.conf' berücksichtigt, welche eindeutig sind. Andere Ansätze aus der Haskell Commnutiy sind, die Versionen generell immer auf die letzten Versionen einzufrieren. Dazu bietet Stackage ein 'cabal.config'-File an, welches für alle verfügbaren Libraries die Versionen eingeschränkt.
Dies hat der Vorteil, dass für einen Zustand die Bibliotheken konsistent sind. Jedoch bringt es den Nachteil, dass beim Update einer Libraries möglicherweise unabhängige Libraries auch aktualisiert werden müssen.

Anhang

Verantwortlichkeiten

Während des gesammten Projekts waren die Verantwortlichkeiten wie folgt geregelt:

- **Kirusanth Poopalasingam**
  - Haskell Framework Evaluation
  - Haskell Testing Environment
  - Supplier Komponente
- **Martin Stypinski**
  - Konzept Evaluator
  - Evaluator Komponente
  - Crawler Komponente
  - Infrastruktur Produktivserver

Protokoll

Es wurden jeweils folgende Punkte besprochen

23.02.15

- Verschiedene Ideen besprochen. Geschaut welche in Haskell einfach oder schwieriger umsetzbar sind.
- Vorgehen besprochen, wie Haskell am effektivsten erlernt werden kann.
- Prof. Dr. Josef. M. Joller hat zwei Vorlesung empfohlen, welche sich gut als Einstieg eignen.

02.03.15

- Fortschritt mit Haskell angeschaut
- Prof. Dr. Josef. M. Joller empfiehlt die Übungen nicht zu machen, da diese meist zu schwierig sind. Ausserdem sind diese zu theoretisch.

09.03.15

- Projektplan
- Architektur Überlegungen
- Ziel: Architekturdurchstich
- Ziel: Evaluator PoC
- Ziel: Infrastruktur organisiert

16.03.15

- Stand Dokumentation
- Infrastruktur
- Zwischenstand: Wikipedia Daten einspielen
- Crawler für 'The Guardian' ausprogrammieren

23.03.15

- Fragen zu Mondas, map, Konzepte
- UI Ideen besprechen
- DB Model anschauen / Performance

- Detailierteres Konzept besprechen
- Risikoanalyse anschauen
- Bericht anschauen
- Implementierung Evaluator alt vs neu
- Benchmarking des Evaluators anschauen
- Crawler
- Wiki Dumper

30.03.15

- Keine nennenswerten Traktanden

13.04.15

- Kirusanth abwesend
- Stand Produktüberlegungen

20.04.15

- Keine nennenswerten Traktanden

04.05.15

- Aufgabenstellung festgehalten
- Fortschritt bei der Dokumentation angeschaut

18.05.15

- Web Frontend gezeigt
- Fortschritt bei der Dokumentation angeschaut

Advanced Haskell Learned

Einführung

Dieses Kapitel fasst Konzepte von Haskell oder GHC zusammen, welche im Beispielcode verwendet werden. Es dient aber auch als Nachschlagewerk während der Entwicklung um neue Erkenntnisse aus der Haskellwelt nachzuschauen.

Haskell Pragmas

Der GHC Compiler bietet die Möglichkeit mit 'Language extensions'[3] die Haskell Syntax zu erweitern. Dies ist vergleichbar mit Macros in C++. Jedoch ist die Liste der Language Extensions vorgegeben. Die Extension werden entweder direkt in der Datei angegeben oder mittels:

```
{-# LANGUAGE ExtensionName #-}
```

Diese können auch direkt in der Cabal-Datei konfiguriert werden.

OverloadedStrings

Mit dem OverloadedStrings können Strings interoperable mit 'Text' und 'ByteString' verwendet werden. Diese müssen nicht expliziet konvertiert werden. Im Hintegrund werden diese sowieso implizit konvertiert.

```
{-# LANGUAGE OverloadedStrings #-}
b :: Text
b = "hello"

-- ohne Extension
import qualified Data.Text as T
c :: Text
c = T.pack "Hello"
```

[3] https://downloads.haskell.org/~ghc/7.4.1/docs/html/libraries/Cabal-1.14.0/Language-Haskell-Extension.html

TemplateHaskell

Template Haskell ist eine Bibliothek und ein Haskell Extension. Sie wird dazu verwendet um zur Compiletime den AST des aktuellen Codes zu manipulieren. Dazusteht folgender 'Operator' [|...|] zu Verfügung.
Beispiel:

```haskell
{-# LANGUAGE TemplateHaskell #-}
-- Expression beinhaltet den Ausdruck als AST ( vom Typ Q Exp )
expression :: Q Exp
expression = [| \x -> x |]

-- definiert den Type vom obigen expression
typeOfExpression :: Q Type
typeOfExpression = [t| String -> String |]

-- Der AST kann auch in Haskell code verwendet werden, dazu muss der AST
↪  wieder in Haskellcode konvertiert werden
-- Dies geschieht mit dem $() ausdruck
usableExpression :: $typeOfExpression
usableExpression :: $expression

-- b == 10
let b = usableExpression 10
```

Quelle: http://edsko.net/2013/05/09/brief-intro-to-quasi-quotation/

QuasiQuotation

QuasiQutoation ist eine Extension welche auf der TemplateHaskell-Extension aufbaut. Mittels QuasiQuotation-Brackets könnnen für beliebiege DSL ASTs erzeugt werden. Im folgenden Beispiel ist dies dargestellt:

```haskell
{-# LANGUAGE TemplateHaskell #-}

-- Wenn wir ein minimales SQL in Haskell modelieren wollten
type Column = String
type Table = String
data Condition = Like Column String
               | Equals Column String
data Select = Select [Column]
data From = From Table
data Where = Where [Condition]
data Sql = Sql Select From Where
           deriving (Data, Typable, Show, Eq)

-- Wenn man nun ein Query formulieren will, würde man das so machen
simpleQuery :: Sql
simpleQuery = Sql (Select ["Person", "Name"]) (From "Person") (Where
↪  [(Like "Person" "Kiru")])

-- Doch eigentlich wäre es lieber direkt sql zu schreiben und man würde
↪  dies zur Compilierzeit zum obigen AST umwandeln
-- Dies ist genau mit QuasiQutation-Extension möglich
-- Dafür muss man einen eine Funktion implementieren, welche den
↪  raw-String nimmt und zu einem AST parst
-- Die Vewendung sieht folgendermassen uas
evenSipmlerQuery :: Sql
evenSipmlerQuery :: [myParser|
                      select Person, Name
                      from Person
                      where Person like "Kiru"
                     ]

-- Man schreibt ein normales SQL und zur Compiletime werden diese zum
↪  richtigen AST umgewandelt, oder falls nicht einen Fehler geworfen.
```

Neben der Verwendung als SQL-Parser könne QuasiQuoter auch verwendet werden, um HTML-Templates direkt in Haskell zu schreiben. Da die Umsetzung eines AST zur Kompilierzeit geschieht, kann der QuasiQuoter so geschrieben werden, dass dieser auf alle verfügbaren Variablen oder Funktionen zugreifen kann.[dV15]

DeriveDataTypeable

Haskell erlaubt für die eigene Typen Instanzen von Eq oder Show automatisch zu erzeugen. Jedoch ist dies auf einige wenige Klassen eingeschränkt. Mit dem DeriveDataTypeable erweitert

GHCI dieses Set um die Klassen Data und Typeable.
Wozu wird Data und Typable verwendet?
Data und Typable sind vergleichbar mit dem reflection Ansatz von Java. Alle Typen vom Data Instanz erlauben es Informationen über die Felder zu holen. Mit dem Typable kann der effektive Type bestimmt werden und von einem "generischen"Wert auf einen bestimmten Type gecastet werden.[4]

Haskell Code Conventions

Für den Haskell Code gelten folgenden Anforderung:

- Keine Operatoren als Funktionsnamen (da sie die Lesbarkeit hindern)
- Lange Funktionen aufsplitten, oder mit dem 'where'-Konstrukt lesbarer machen
- Unabhängige Komponenten in Module aufteilen, welche unabhängig getestet werden können
- Für die Imports der Module gelten folgenden Regeln:

```
import Data.List -- nicht verwenden
-- immer qualified importieren, wenn alle Methoden eingebunden
↪  werden
-- Warum? Es vereinfacht das Auffinden von methoden.
import qualified Data.List as D
-- Alternative, wenn die einzelnen Methoden implementieren
import Data.List ( genericIndex )
```

[4] http://chrisdone.com/posts/data-typeable

Abbildungsverzeichnis

Glossar

Arrows

Generalisierung von den Monaden. Erlaubt es Funktionen in einem Point-free-style zu schreiben ($f \cdot g \cdot s$), auch wenn manche davon mit Seiteneffekten belastet sind. 8

Bayes-Klassifikator

Ein Verfahren nach Thomas Bayes, welches es ermöglicht beliebige Objekte in Gruppen einzuteilen. Beispiel wäre hier der Spam-Filter. 6

Cabal

Standard Package Manager für Haskell. Ähnlich wie SBT, Maven oder Ant. 47

Clusteranalyse

Ein Verfahren zur Feststellung von Ähnlichkeiten in Datenbeständen. 6

GHC

Glasgow Haskell Compiler ist ein verbreiteter Compiler für Haskell. 47

Haskell

An advanced purely-functional programming language. 4, 14, 15

Monade

Eine Abstraktion in Haskell um die Seiteneffekte zu verstecken / typisieren. 8

PostgreSQL

PostgreSQL ist ein freies, objektrelationales Datenbanksystem. Seit 1997 wird das Produkt hauptsächlich von OpenSource Communities gepflegt und weiterentwickelt. 14

RDBMS

Relationales Datenbankmanagementsystem (PostgreSQL, mySQL, Oracle Database, db2). 14

Webstack

Ein 'Full Web Stack' umschreibt alle Komponenten die benötigt werden um eine Webseite auszuliefern - von der Datenhaltung bis zum Webserver. 15

Literaturverzeichnis

[All15] Chris Allen. Introduction to haskell. https://github.com/bitemyapp/learnhaskell#primary-courses, 2015. [Online; 2015-04-06].

[Bha15] Aditya Bhargava. Making a website with haskell. http://adit.io/posts/2013-04-15-making-a-website-with-haskell.html, 2015. [Online; 2015-03-28].

[dV15] Edsko de Vries. Quasi-quoting dsls for free. http://www.well-typed.com/blog/2014/10/ttp://hspec.github.io/quasi-quoting-dsls/, 2015. [Online; 2015-06-02].

[Gra15] Paul Graham. A plan for spam. http://www.paulgraham.com/spam.html, 2015. [Online; 2015-04-03].

[hap15] Happstack crashcourse. http://happstack.com/docs/crashcourse/index.html, 2015. [Online; 2015-05-20].

[Hut07] Graham Hutton. *Programming in Haskell*. Cambridge University Press, 2007.

[KN02] D. Stoffer K. Nipp. *Lineare Algebra*. vdf, 5 edition, 2002.

[Lip11] Miran Lipovaca. *Learn You a Haskell for Great Good!: A Beginner's Guide*. No Starch Press, 2011. http://learnyouahaskell.com/chapters.

[Mil15] Ivan Miljenovic. Repeat after me: "cabal is not a package manager". https://ivanmiljenovic.wordpress.com/2010/03/15/repeat-after-me-cabal-is-not-a-package-manager/, 2015. [Online; 2015-03-28].

[Mü15] Prof Dr. Andreas Müller. Wahrscheinlichkeitsrechung und statistik. https://github.com/AndreasFMueller/WrStat, 2015. [Online; 2015-04-03].

[sna15] Snaplets tutorial. http://snapframework.com/docs/tutorials/snaplets-tutorial, 2015. [Online; 2015-03-28].

[Sno12] Michael Snoyman. *Developing Web Applications with Haskell and Yesod*. O'Reilly Media, 2012.

[Yor15] Brent Yorgey. Introduction to haskell. http://www.seas.upenn.edu/~cis194/spring13/lectures.html, 2015. [Online; 2015-04-06].