Eine kontextsensitive Auto-Vervollständigung für webbasierte Anwendungen

Bachelorarbeit

von

Frederik Grieshaber

aus
Solingen

vorgelegt am

Lehrstuhl für Rechnernetze und Kommunikationssysteme
Prof. Dr. Martin Mauve
Heinrich-Heine-Universität Düsseldorf

Juli 2017

Betreuer:
Tobias Krauthoff M.Sc.

Zusammenfassung

Am Lehrstuhl für Technik sozialer Netze der Heinrich-Heine-Universität Düsseldorf wurde im Rahmen des NRW-Fortschrittskolleg *Online-Partizipation* ein neuartiger Ansatz für Diskussionen im Internet entwickelt. Dieser soll die Meinungsfindung erleichtern, indem anstelle einer offenen Foren-Diskussion ein simulierter Dialog eingesetzt wird. Nutzer können hier Aussagen anderer Nutzer auswählen und anhand bereits vorhandener Argumente an der Diskussion teilnehmen. Es besteht allerdings auch die Möglichkeit, an einigen Stellen eigene Aussagen und Gründe einzugeben. Wichtig ist dabei, dass die Eingabe von Duplikaten möglichst verhindert wird. Dafür soll eine Auto-Vervollständigung eingesetzt werden, die abhängig von der aktuellen Situation und der Eingabe des Nutzers optimale Vorschläge ermittelt.

Das Ergebnis dieser Arbeit ist ein Mikroservice, der mit Hilfe eines Elasticsearch-Suchservers bei allen Eingabenmöglichkeiten in D-BAS die Vorschlagsermittlung für die Auto-Vervollständigung durchführt. Die resultierende Lösung arbeitet äußerst effizient und ist wesentlich performanter als der bisherige Ansatz und eine zwischenzeitlich angestrebte Lösung durch phonetische Algorithmen. Für die Evaluation wurde gezeigt, dass die situationsbedingte Priorisierung von Statements korrekt funktioniert. Außerdem wurde die Zeit, die die drei Ansätze (bisherige Umsetzung, Phonetik und Elasticsearch-Lösung) benötigen, um für bestimmte Eingaben die Vorschlagsliste zu ermitteln, gemessen und verglichen. Dabei wurde festgestellt, dass die finale Umsetzung am verlässlichsten und schnellsten Ergebnisse ermittelt.

Inhaltsverzeichnis

Abbildungsverzeichnis **vii**

Tabellenverzeichnis **ix**

Quelltextverzeichnis **xi**

1 Einleitung **1**
- 1.1 Problemstellung 2
- 1.2 Aufbau 2

2 Auto-Vervollständigungen **3**
- 2.1 Google Suchmaschine 3
  - 2.1.1 Google Suggest 4
  - 2.1.2 Google Instant 5
- 2.2 Art der entwickelten Auto-Vervollständigung 5

3 Grundlagen **7**
- 3.1 Phonetische Suche 8
  - 3.1.1 Kölner Phonetik 8
  - 3.1.2 Double Metaphone 10
- 3.2 Elasticsearch 11
  - 3.2.1 Aufbau und Datenspeicherung 11
  - 3.2.2 Document und Type 11
  - 3.2.3 Scoring 12
  - 3.2.4 Suche 13
  - 3.2.5 Analyzer und Filter 13

4 Implementierung **15**
- 4.1 Hauptprogramm 15
  - 4.1.1 Suchmöglichkeiten und Schnittstellen 16
  - 4.1.2 Ansatz mit phonetischen Algorithmen I 20

4.2 Elasticsearch-Server 20
4.2.1 Autocomplete-Analyzer 21
4.2.2 Suchanfragen 23

5 Evaluation **27**
5.1 Bisherige Vorgehensweise 27
5.2 Ansatz mit phonetischen Algorithmen II 28
5.3 Bewertung der Kontextsensitivität 29
5.3.1 Speicherung eines Argumentes in D-BAS 29
5.3.2 Suchmöglichkeiten aus Sicht von D-BAS 30
5.4 Vergleich Elasticsearch vs. Phonetik vs. bisherigem Ansatz 37
5.4.1 Performance 37
5.4.2 Qualtität der Ergebnisse 40
5.5 Verbesserungsmöglichkeiten 40

6 Zusammenfassung und Ausblick **43**
6.1 Zusammenfassung 43
6.2 Ausblick 44

Literaturverzeichnis **45**

Abbildungsverzeichnis

2.1 Ergänzung einer ganzer Frage von Suggest 4
2.2 Screenshot der Google-Suche 4

4.1 Screenshot von D-BAS bei Suche in allen Aussagen 16
4.2 Screenshot von D-BAS bei Suche in allen Aussagen 16
4.3 Screenshot von D-BAS bei Eingabe einer neuen Position 17
4.4 Screenshot von D-BAS bei einfacher Begründung 18
4.5 Screenshot von D-BAS bei Begründung eines Arguments 19

5.1 Vergleich Suchzeit der Algorithmen nach Wortlänge 38
5.2 Vergleich Suchzeit der Algorithmen nach Anzahl Wörtern 39

Tabellenverzeichnis

3.1 Regeln der Kölner Phonetik 9
3.2 Beispiel für Kölner Phonetik 10

4.1 Beispiel edge_ngram-Filter 22
4.2 Anzahl erlaubter Änderungsoperationen nach Wortlänge 24

5.1 Kriterien für aktuelle Suchfunktion in D-BAS 28

Quelltextverzeichnis

4.1 Mapping für Statements 21

4.2 Erstellen des Analyzers 22

4.3 Beispiel einer einfachen Suchanfrage 23

4.4 Beispiel einer Suchanfrage für Start-Positionen 25

4.5 Beispiel einer Suchanfrage mit gegebenen IDs 26

5.6 Ausschnitt Suchanfrage im ersten Fall 32

5.7 Ausschnitt aus Ergebnis im ersten Fall 33

5.8 Ausschnitt Suchanfrage im zweiten Fall 1 34

5.9 Ausschnitt aus Ergebnis im zweiten Fall 1 35

5.10 Ausschnitt aus Suchanfrage im zweiten Fall 2 36

5.11 Ausschnitt aus Ergebnis im zweiten Fall 2 37

Kapitel 1

Einleitung

Am Lehrstuhl für Technik Sozialer Netzwerke der Heinrich-Heine-Universität Düsseldorf wird ein dialogbasiertes Argumentationssystem (kurz: D-BAS) entwickelt [KBBM16], dessen Ziel die Entwicklung einer neuartigen Form der Online-Argumentation ist. Man entfernt sich von unstrukturierten Systemen, wie einfachen Foren oder komplexen Ansätzen, wie Argumentationskarten. Stattdessen wird der Nutzer bzw. die Nutzerin[1] durch einen simulierten Dialog geführt. Dadurch soll eine Diskussionsform geschaffen werden, durch die die Diskussionen auf die Kernaussagen reduziert werden können und eine Meinungsfindung wesentlich erleichtert werden soll. Zu Beginn der Diskussion wählt der Nutzer ein Thema aus, über das er reden möchte. Daraufhin werden ihm verschiedene Startpunkte präsentiert. Nachdem er eine dieser Positionen ausgewählt hat, wird er gebeten, dazu Stellung zu nehmen. Dafür kann der Nutzer der Aussage zustimmen oder sie ablehnen und muss anschließend einen Grund für diese Entscheidung angeben. In der weiteren Diskussion wird der Nutzer immer wieder mit den Meinungen anderer Nutzern konfrontiert, die er wiederrum unter Angabe seiner Gründe bewerten soll.

Das System ist dabei so aufgebaut, dass der Nutzer unter dem Verlauf der Diskussion einige Auswahlmöglichkeiten für die aktuelle Situation präsentiert bekommt. Die Reaktionsmöglichkeiten auf Argumente bzw. Aussagen sind dabei immer statisch, während die Gründe durch die Eingaben der anderen Teilnehmer repräsentiert werden. Im Rahmen einer weiteren Bachelorarbeit am Lehrstuhl wurde die Möglichkeit eines Recommender Systems erforscht [Gen17], welches eine begrenzte Anzahl zum Nutzerverhalten passender Gründe vorschlägt und somit die Übersichtlichkeit der Auswahl verbessert. Es bestimmt jedoch weiterhin die Möglichkeit sich alle alle Gründe (*Prämissen*) zur aktuellen Aussage (*Konklusion*) anzeigen zu lassen. Anstatt sich für eine der Auswahlmöglichkeiten zu entscheiden, hat der Nutzer auch die Möglichkeit, eigene Aussagen einzugeben.

[1] Aus Gründen der Lesbarkeit dieser Arbeit wird im folgenden ausschließlich *Nutzer* verwendet. Darin sind jedoch Personen jeglichen Geschlechts inbegriffen.

1.1 Problemstellung

Es ist davon auszugehen, dass Nutzer in einer großen Diskussion mit vielen Aussagen nicht erst alle Prämissen durchlesen und anschauen wird. Sollte keine der vorgeschlagenen Aussagen der Meinung des Nutzers entsprechen, so wird dieser geneigt sein, seine eigene Aussage einzugeben. Damit es hierbei nicht zu Duplikaten kommt, soll in dieser Arbeit ein Mikroservice entwickelt werden, der kontextsensitiv auf die Eingaben des Nutzers reagiert und dazu passende Einträge vorschlägt. Dies soll passieren, während der Nutzer tippt - die Auto-Vervollständigung hat also eine *search-as-you-type* Funktionalität.

Um das zu erreichen, müssen alle Aussagen (*statements*) ermittelt werden, die den eingegebenen oder dazu ähnlichen Text enthalten. Wichtig ist hierbei, dass mögliche Rechtschreibe- und Tippfehler, sowie ein anderer Satzbau das Ergebnis nicht negativ beeinflussen. Außerdem sollte das erarbeitete Verfahren effizient und schnell arbeiten, um eine positive User-Experience zu ermöglichen.

Das Ziel dieser Arbeit ist es also, eine Auto-Vervollständigung zu entwickeln, die möglichst viele Statements für das Ergebnis berücksichtigt (also *das Netz sehr weit spannt*), dabei jedoch darauf achtet, dass jedes Ergebnis für den Nutzer nachvollziehbar ist. Desweiteren sollten die höher priorisierten Treffer möglichst einen erkennbaren Bezug zur aktuellen Situationen aufweisen.

1.2 Aufbau

Im Anschluss an die Einleitung wird ein Überblick über den Einsatz von Auto-Vervollständigungen vermittelt. Anschließend werden einige Grundlagen erklärt, um das technische und theoretische Verständnis der zum Einsatz gekommener Techniken zu verbessern. Im darauffolgenden Kapitel wird die Umsetzung des entwickelten Mikroservices und den darin vorkommenden Komponenten vorgestellt. Dann folgt die Evaluation, in der verschiedene Ansätze miteinander verglichen werden. Dabei werden die Laufzeit und die Qualität der Ergebnisse ausgewertet. Den Schluss bildet eine Zusammenfassund der Arbeit und ein Ausblick bezogen auf diese Arbeit, aber auch auf Auto-Vervollständigungen im Allgemeinen.

Kapitel 2

Auto-Vervollständigungen

Auto-Vervollständigungen kommen mittlerweile an sehr vielen Stellen, an denen irgendeine Form der Texteingabe stattfindet, als Komfortfunktion zum Einsatz. Das kann die Vorschlagsliste beim Tippen von Nachrichten auf Smartphones sein, wo zum einen Tippfehler behoben und auch schon nächste Wörter vorgeschlagen werden, die der Nutzer häufig in dieser Reihenfolge benutzt hat. Das kann aber auch die automatische Code-Vervollständigung in Entwicklungsumgebungen wie Intellij oder Eclipse sein, bei der zum Beispiel vorher deklarierte Variablen und Methoden als Kontext mit einbezogen und passend vorgeschlagen werden.

Die wohl bekannteste und wegweisendste Umsetzung einer Auto-Vervollständigung ist bei der *Google*-Suchmaschine zu finden und wird in diesem Kapitel vorgestellt. Außerdem wird beschrieben, wie sich dieser Arbeit in die vorgestellten System eingliedert.

2.1 Google Suchmaschine

Mit dem Ziel alle Informationen dieser Welt zu ordnen und erreichbar zu machen, gründen Larry Page und Sergey Brin 1997 das Unternehmen *Google* [goo17b]. Im selben Jahr wurde die gleichbenannte Suchmaschine veröffentlicht. Seitdem wächst sie stetig und wird regelmäßig mit neuen Features, wie der Bildersuche (2001) oder Google Shopping (2002, damals: *Froggle*) erweitert. 2014 befanden sich etwa 30 Billionen Dokumente in Googles Index [goo17c], wobei die aktuelle Zahl deutlich größer sein sollte (von 2013 bis 2014 wuchs der Index allein um 8 Billionen Dokumente).

2.1.1 Google Suggest

Seit 2008 wird *Google Suggest* [Höl08] für die offizielle Google-Suchmaschine eingesetzt. *Suggest* ist das bekannteste Feature, dass bereits während des Tippens der Suchanfrage Vorschläge für den Suchtext ermittelt und als Liste unter dem Suchfeld anzeigt. Ursprünglich wurden die Vorschläge lediglich durch die Anfragestatistiken aller Nutzer ermittelt. Mittlerweile werden weitere Faktoren für die Vorschlagsermittlung ausgewertet, wie zum Beispiel der Ort und vorherige Anfragen des Nutzers, sowie aktuelle Trendthemen [goo17a].

Bei der Suche wird eine begrenzte Anzahl an Rechtschreibfehler toleriert. Die Vorschläge, die *Suggest* ermittelt, reichen von einzelnen Worten bis hin zu ganzen Phrasen. Ausschlaggebend ist dabei ist die Art des eingebenen Wortes. Wird zum Beispiel ein Fragewort eingegeben, so werden ganze Fragen ergänzt und vorgeschlagen.

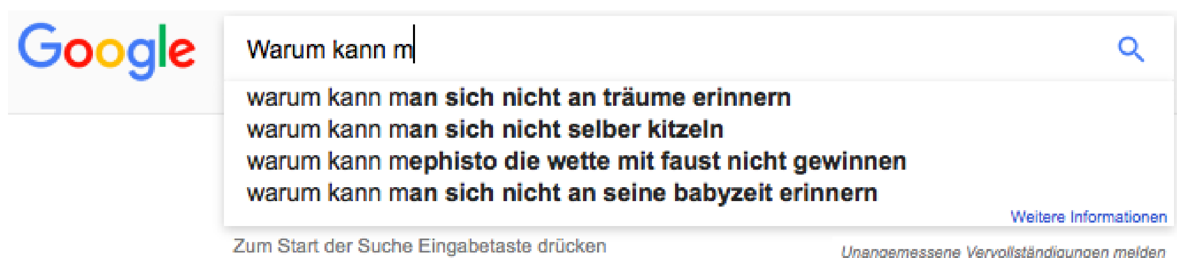

Abbildung 2.1: Ergänzung einer ganzer Frage von Suggest

Bei selteneren oder komplexeren Wörtern beinhaltet die Vorschlagsliste anfangs nur einzelne Wörter, bis dieses ausreichend lang eingegeben wurde. Dann versucht *Suggest* häufige Wortkombinationen zu ermitteln. Werden mehrere Wörter eingegeben, die in dieser Kombination nicht bekannt oder nicht populär sind, dann wird das aktuell getippte Wort für sich allein betrachtet.

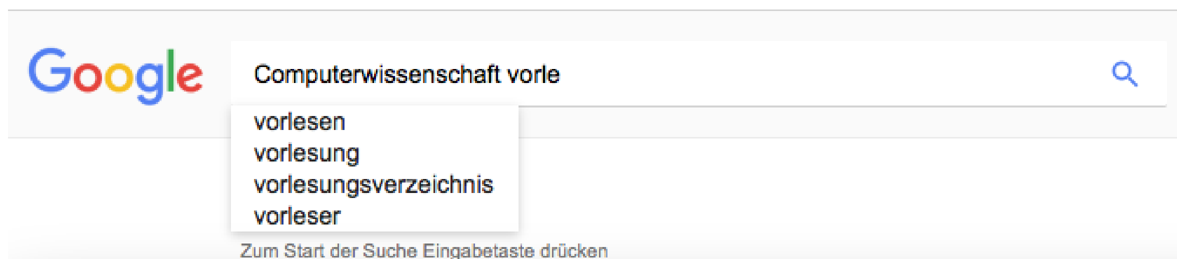

Abbildung 2.2: Screenshot der Google-Suche

Kritik an Google Suggest

Der ausschlaggebenste Faktor für die Vorschläge ist das Suchvolumen, also wie oft tatsächlich nach bestimmten Begriffen gesucht wurde. In Deutschland erregte diese Methode 2012 Aufsehen, als Bettina Wulff, Ex-Frau des ehemaligen Bundespräsidenten der BRD Christian Wulff, in ingesamt 43 Wortkombinationen mit dem Rotlichtmilieu in Verbindung gebracht wurde (vgl. [Wul15]) und Google deswegen verklagte. Trotz der urprünglichen Aussage, die Suchergebnisse nicht verfälschen zu wollen, wurden 2015 die entsprechden Kombinationen gelöscht und die Richtlinien angepasst, sodass künftig persönlichkeitsverletzende Aussagen unterbunden wurden.

Google äußert sich klar für freie Meinungsäußerung und freien Informationsfluss, verhindert aber anhand einiger Richtlinien, dass bestimmte Inhalte in der Vorschlagsliste bzw. in den Suchergebnissen vorkommen. Diese sind zum Teil moralisch begründet, wie zum Beispiel das Entfernen von kinderpornographischen Inhalten, oder unterliegen gesetzlichen Vorschriften in den jeweiligen Ländern. So werden beispielsweise in Deutschland Nationalsozialimus-verherrlichende Inhalte entfernt.

2.1.2 Google Instant

Zwei Jahre nach der offiziellen Einführung von *Suggest* wurde 2010 *Google Instant* [goo17d] veröffentlicht. In Studien, die unter anderem mit dem *eye-tracking* Verfahren arbeiteten, wurde festgestellt, dass ein Nutzer wesentlich schneller auf andere Teile der Website schaut, als dass er weitere Buchstaben tippt. Mit *Instant* werden während des Tippens nicht nur Vorschläge für die Suchanfrage ermittelt, sondern direkt Ergebnisse gesucht und angezeigt. Dadurch soll die Zeit, die ein Nutzer für eine Suche benötigt, reduziert werden.

2.2 Art der entwickelten Auto-Vervollständigung

Alle oben aufgezeigten Varianten einer Auto-Vervollständigung dienen ausschließlich als Komfortfunktion für Benutzer. Durch das Vorschlagen ermittelter Wörter oder Texte soll der Nutzer mit diesen Varianten Zeit sparen, in dem er die Eingabe nicht vollenden muss, sondern sich vorher bereits für einen Vorschlag entscheidet.

Bei der Google-Suchmaschine ist es außerdem so, dass die Vorschläge unabhängig vom Ergebnis der Suche sind. Beispielsweise können bei unbekannten Wortkombinationen, bei denen die Wörter einzeln betrachtet und vervollständigt werden, Vorschläge erscheinen, die in keinem Ergebnis vorhanden sind.

Die in dieser Arbeit entwickelte Auto-Vervollständigung hat andere Anforderungen. Die Hauptaufgabe ist es, einen Überblick über vorhandene Texte zu bieten, damit die Eingabe von Duplikaten vermieden wird. Daher ist es wichtig, dass in der Ergebnisliste bereits die echten Ergebnisse der Suche stehen sollen und nicht erst mögliche Phrasen, mit denen dann erst Ergebnisse ermittelt werden.

Kapitel 3

Grundlagen

Bevor das Ergebnis dieser Arbeit beschrieben und evaluiert werden kann, werden einige technische und theoretische Grundlagen erläutert. Dazu gehört unter anderem ein Überblick über die Phonetik und die phonetische Suche, sowie eine allgemeine Einführung in Elasticsearch [Ela17]. Außerdem wird definiert, wie und in welchem Umfang der Kontext in einer Diskussion für die Vorschläge ausgewertet wird (*Kontextsensitivität*).

Kontextsensitivität

Die in dieser Arbeit entwickelte Auto-Vervollständigung soll kontextsensitiv agieren. Das heißt, sie soll nicht alle Statements gleichbewertend durchsuchen und Treffer nur anhand der Eingabe ermitteln, sondern die aktuelle Situation des Nutzers mit einbeziehen und auswerten. Mit Hilfe dieser Informationen soll dann versucht werden, einige Aussagen zu priorisieren bzw. zu bevorzugen.

In den ersten Anläufen einer Umsetzung wurde dazu versucht, das Nutzerverhalten verstärkt zu analysieren und die *Meinung*[1] des Nutzers bezüglich des gewählten Themas zu bestimmen. Dabei hat sich jedoch rausgestellt, dass in den Situationen, in denen eine Eingabe möglich ist, diese Meinung nahezu irrelevant ist. D-BAS ist so konzipiert, dass Diskutanten entweder eine einzelne Aussage begründen oder Argumente anderer Nutzer bewerten sollen. Dabei ist die Information, wie der Nutzer sich ursprünglich zum Thema geäußert hat, für die Priorisierung der Vorschläge nicht nützlich.

Stattdessen ist davon auszugehen, dass Statements, die den von D-BAS vorgegebenen Satzanfang (zum Beispiel *Was ist ihr wichtigster Grund dafür, dass...*) sinnvoll ergänzen würden, ebenfalls der Stimmung des Nutzers entsprechen würden. Daher haben solche Ergebnisse einen höheren *score*, also eine höhere Relevanz für das Ergebnis, und werden somit weiter oben in der Ergebnisliste angezeigt.

[1] Neben der Teilnahme an einer Diskussion getreu der eigenen Meinung, ist es durchaus denkbar, dass Nutzer nur aus Interesse andere Statements anklicken. Für diese Arbeit wird das Verhalten in beiden Szenarien als Meinung des Nutzers bezeichnet.

Allerdings ist es nicht ausreichend und sinnvoll, nur die Statements zu durchsuchen, die in die oben beschriebene Situation passen. Da es hauptsächlich um die Vermeidung von Duplikaten geht, werden alle Statements durchsucht. Somit können auch solche Aussagen vorgeschlagen werden, die in ganz anderen Diskussionen oder Konstellationen vorkamen.

3.1 Phonetische Suche

> Die Phonetik ist ein Teilgebiet der Linguistik, das sich mit der Erzeugung von sprachlichen Lauten und deren Verwendung für die Kommunikation beschäftigt [Dud].

Die phonetische Suche bietet die Möglichkeit zum Beispiel Rechtschreibvarianten eines Wortes zu erkennen und somit das Suchergebnis zu verbessern. Bei den zugrunde liegenden phonetischen Algorithmen werden für Wörter repräsentative, phonetische Codes ermittelt, mit denen man diese dann miteinander vergleichen kann. Ist der phonetische Code zweier Wörter gleich, so sollen die Wörter einen ähnlichen Wortlaut haben. Mit dieser Methode lassen sich beispielsweise alle Derivate des Namens *Meier*, etwa *Meyer* oder *Maier* finden. Da in D-BAS zurzeit in Deutsch und Englisch diskutiert werden kann und phonetische Algorithmen meist auf eine Sprache angepasst sind, wurden für diese Arbeit zwei Algorithmen ausgewählt. Diese werden im Folgenden vorgestellt.

3.1.1 Kölner Phonetik

Für die deutsche Sprache wurde der *Kölner Phonetik* - Algorithmus [Pos69] ausgewählt. Er wurde 1969 von Hans Joachim Postel veröffentlicht und ist eine Anpassung des *Soundex* - Algorithmus [OR18] an die deutsche Sprache.

Es wird für jeden Buchstaben eines Wortes eine Ziffer zwischen 0 und 8 ermittelt, wobei maximal ein benachbarter Buchstabe als Kontext mit einfließt (vgl. Tabelle 3.1).

Nr.	Buchstabe	Kontext	Code
1	A, E, I, J, O, U, Y	–	0
2	H	–	–
3	B	–	1
4	P	außer vor H	1
5	D,T	außer vor C, S, Z	2
6	F, V, W	–	3
7	P	vor H	3
8	G, K, Q	–	4
9	C	am Wortbeginn vor A, H, K, L, O, Q, R, U, X	4
10	C	vor A, H, K, L, O, Q, R, U, X, aber nicht nach S, Z	4
11	X	außer nach C, K, Q	48
12	L	–	5
13	M, N	–	6
14	R	–	7
15	S, Z	–	8
16	C	nach S, Z	8
17	C	am Wortbeginn, außer vor A, H, K, L, O, Q, R, U, X	8
18	C	außer vor A, H, K, L, O, Q, R, U, X	8
19	D, T	vor C, S, Z	8
20	X	nach C, K, Q	8

Tabelle 3.1: Regeln der Kölner Phonetik

Nachdem der phonetische Code bestimmt wurde, werden alle mehrfach nebeneinander vorkommenenden Ziffern, sowie anschließend alle nicht am Anfang stehende Nullen entfernt.

Beispiel 3.1.1.1 Bestimmung des phonetischen Codes des Wortes *Informatik* nach Abschnitt 3.1.1
Als erstes werden die Buchstaben gemäß der Regeln aus Tabelle 3.1 durch Ziffern ersetzt:

Zeichen	Code	Regel
I	0	1
N	6	13
F	3	6
O	0	1
R	7	14
A	0	1
T	2	5
I	0	1
K	4	8

Tabelle 3.2: Beispiel für Kölner Phonetik

Dadurch entsteht die Ziffernfolge 063070204, aus der jetzt alle nicht-führenden Nullen entfernt werden. Man erhält so den Code 063724 für das Wort Informatik nach der Kölner Phonetik, da keine Ziffern nebeneinander mehrfach vorkommen und somit nichts mehr geändert werden muss.

3.1.2 Double Metaphone

Für die englische Sprache wurde der *Double Metaphone* - Algorithmus [Phi00] gewählt. Im Jahr 2000 wurde er von Lawrence Philips als die Weiterentwicklung des *Metaphone* - Algorithmus [Phi90] veröffentlicht.

Anders als bei der *Kölner Phonetik* werden hier die Buchstaben nicht pauschal durch Ziffern ersetzt. Die Wörter werden anhand eines komplexen Regelwerks so transformiert, dass Buchstaben in bestimmten Kontexten zu anderen Buchstaben oder Ziffern werden (z.B. *ch* wird zu *k*) oder wegfallen. Es werden außerdem für jedes Wort zwei Codes ermittelt, sodass die Ähnlichkeit abgestuft werden kann. Die Wörter ähneln sich sehr, wenn ihre beiden primären Codes gleich sind; weniger, wenn der primäre Code des einen Wortes mit dem sekundären des anderen übereinstimmt und am wenigsten, wenn nur die beiden sekundären identisch sind.

3.2 Elasticsearch

Elasticsearch ist eine Open-Source Such- und Analysemaschine basierend auf der Lucene Suchmaschine. Durch den Einsatz einer RESTful-Schnittstelle soll die Suche einfach und vorallem benutzerfreundlich gestaltet werden. Außerdem ist sie dadurch auf vielen Wegen einsetzbar. Sie kann über die Elastic-Applikation Kibana [Kib17] im Web-Browser gesteuert werden, aus der Kommando-Zeile oder aus vielen Programmiersprachen. Neben der reinen Volltext-Suche bietet Elasticsearch außerdem Schnittstellen an, um seine Daten zu analysieren, anzureichern und zu überwachen [GT15].

3.2.1 Aufbau und Datenspeicherung

Die Software wird auf mindestens einem Server, genannt *node*, installiert. Um ständige Erreichbarkeit und gute Skalierbarkeit zu realisieren und Datenverlust vorzubeugen, werden häufig mehrere Nodes zu einem *cluster* zusammengeschlossen. Der Zusammenschluss, die Datenorganisation etc. werden hier automatisch von Elasticsearch übernommen, sofern in jeder *node* - Konfiguration der selbe Clustername eingetragen ist.

Ähnliche bzw. zusammenhängende Daten werden in *indices* gespeichert. Um ein hohes Maß der Flexibilität bei der Speicherung und Verwaltung der Daten zu erhalten, werden diese als JSON-Objekte abgespeichert.

3.2.2 Document und Type

Ein *type* würde in einer relationalen Datenbank einer Tabelle entsprechen. Er beschreibt die Felder (*properties*) der gespeicherten Daten (*documents*). Anders jedoch als bei der relationalen Speicherung, sollten die Daten hier möglichst denormalisiert sein. Jedes Objekt sollte also alle Informationen beinhalten, die es für die Suche oder Analyse braucht. Sofern nicht anders angegeben, werden alle Felder standardmäßig indiziert und sind damit durchsuchbar. Man kann außerdem jedem Feld einen Datentyp zuweisen, wie zum Beispiel *text* und *date* oder komplexeren, wie Geo-Informationen. Mit jedem Datentyp gehen verschiedene Eigenschaften einher, die auf unterschiedliche Arten analysiert werden können.

3.2.3 Scoring

Jedes Ergebnis (*hit*) einer Suchanfrage enthält, neben den eigentlichen Inhalten, weitere Metainformationen. Dazu gehört der *score*. Diese positive, reelle Zahl dient als Maß der Passgenauigkeit bzw. der Relevanz des Ergebnisses und wird benutzt, um die Dokumente zu sortieren. Die Berechnung des *scores* basiert meist auf den folgenden Mechanismen:

Term frequenzy

Dieser Faktor ermittelt sich durch die Anzahl der Vorkommnisse des Suchtextes [2] im durchsuchten Dokument. Je häufiger er vorkommt, umso wahrscheinlicher ist es, dass das Dokument für die Suche relevanter ist.

Inverse document frequenzy

Bei der Berechnung dieses Faktors spielt die Anzahl der Vorkommnisse des Suchtextes im gesamten Index die entscheidene Rolle. Je häufiger die Phrase im Index vorkommt, desto geringer wird seine Relevanz sein. Ein Wort, das in jedem Dokument vorkommt, ist folglich weniger ausschlaggebend für das Ergebnis, als eins, das nur in einem Dokument vorkommt.

Field-length norm

Dieser Faktor setzt die oben ermittelten Häufigkeiten in Relation zur Länge des Feldes. Kommt ein Wort oder eine Phrase in einem kurzen Text vor, so bekommt er eine höhere Relevanz zugeordnet, als würde er in einem langen Text vorkommen.

Neben diesen Berechnungen kann auch die Art der Suchanfrage Einfluss auf den *score* haben. Benutzt man zum Beispiel eine *fuzzy-query*, also eine Suchanfrage, die anhand der Levensthein-Distanz ähnliche Suchbegriffe bestimmt, um Rechtschreibfehler zu tolerieren, so wird der *score* reduziert, falls das Dokument nur nach einer Korrektur des Suchtextes gefunden werden konnte. Neben den Standardeinstellungen für die Ermittlung des *scores* besteht die Möglichkeit, eigene Berechnungsvorschriften zu erstellen.

[2] Suchtexte werden in der aktuellen Konfiguration in ihre Wörter zerlegt; daher müssen die hier stehenden Aussagen für jedes Wort betrachtet werden

Levensthein-Distanz

Die Levensthein-Distanz zwischen zwei Wörtern ist die minimale Anzahl an Änderungen (einfügen, löschen und vertauschen), die vorgenommen müssen, um das eine Wort in das andere zu überführen.

3.2.4 Suche

Eine Suchanfrage (*query*) kann neben dem eigentlich zu suchenden Text weitere Filter und Einschränkungen enthalten. Dafür können Klauseln definiert werden, die durch *bool-query* beliebig miteinander verbunden werden können. Es gibt vier verschiedene Arten der Klauseln, die beschreiben, wie sich die Bedingungen auf den *score* auswirken und wie strikt sie sind. Dafür werden die vier Schlüsselwörter **must, filter, should** und **must_not** verwendet. Eine **must** - Klausel besagt, dass jedes Ergebnis die definierte Bedingungen erfüllen muss. Außerdem beeinflussen die Bedingungen den *score*. Dazu ähnlich ist die **filter** - Klausel, bei der die Bedingungen ebenfalls auf alle Ergebnisse zutreffen müssen, jedoch keinen Einfluss auf den *score* haben. Das Gegenteil dazu bietet die **must_not** - Klausel. Hier definierte Bedingungen sollen auf kein Ergebnis zutreffen. Alle Dokumente, die eine der definierten Anforderungen erfüllen, bekommen einen *score* von 0. Mit einer **should** - Klausel können mehrere Bedinungen definiert werden, die aber nicht alle auf die Ergebnisse zutreffen müssen. Über den zusätzlichen Parameter *minimum_should_match* kann angegeben werden, wie viele der beschriebenen Anforderungen auf die Ergebnisse mindestens zutreffen müssen. Außerdem wird durch diese Klausel der *score* beeinflusst, sodass Dokumente, die mehrere Bedinungen erfüllen entsprechend höher bewertet werden.

3.2.5 Analyzer und Filter

Es besteht die Möglichkeit, bei der Indizierung (also beim Speichern neuer Dokumente) oder zum Zeitpunkt der Suche, vorher definierte Analyzer und Filter auf die Felder des Dokuments anzuwenden. Beide Zeitpunkte haben unterschiedliche Vor- und Nachteile. Die Analyse zur Indizierung hat einen größeren Index (mehr Speicherbedarf) zur Folge, bietet jedoch deutlich performantere Suchanfragen. Dazu müssen sogenannte *mappings* definiert werden. Diese weisen den Feldern des Dokuments die gewünschten Filter und Analyzer zu. So können unterschiedliche Felder mit verschiedenen Methoden analysiert werden. Es besteht außerdem die Möglichkeit, für die Indizierung und die Suche unterschiedliche Analyzer zu zuweisen.

Für die Analyse zur Suchzeit werden keine *mappings* definiert, was dazu führt, dass die Suchanfragen länger dauern, da die Analyse erst dann durchgeführt wird. Die Vorteile dieses Zeitpunktes sind ein

geringerer Speicherbedarf und höhere Flexibilität. Letzteres resultiert daraus, dass falls neue *mappings* definiert werden, die gesamte Datenmenge neu indiziert werden muss.

Es gibt zum Beispiel unter anderem die *Tokenizer* (zerlegen der Texte nach bestimmten Kriterien), *lowercase*-Filter (alles in Kleinbuchstaben überführen), *stopword*-Filter (je nach Sprache Artikel etc. entfernen), *n-Gram*-Filter (zerlegen der Token in alle möglichen Teile mit einer definierten Längen) und viele weitere.

Es besteht dazu die Möglichkeit eigene Analyzer bzw. Filter zu konfigurieren, indem man die oben genannten miteinander verknüpft oder kombiniert. So ensteht der für diese Arbeit wichtige *autocomplete*-Analyzer, der im nachfolgenden Kapitel genauer erklärt wird.

Kapitel 4

Implementierung

In D-BAS gibt es mehrere Eingabemöglichkeiten, bei denen die Auto-Vervollständigung zum Einsatz kommen soll. Daher wurden in dem entwickelten Mikroservice sechs Schnittstellen implementiert, die jeweils auf eine der Möglichkeiten angepasst sind. In diesem Kapitel wird beschrieben, wie die Situationen im Detail aussehen und wie die zugehörigen Schnittstellen arbeiten. Dafür werden zuerst die Situationen erläutert und dann wie die entstehenden Suchanfragen aussehen und wie diese vom Elasticsearch-Server verarbeitet werden. Desweiteren wird die Konfiguration des Servers und wieso sie so gewählt wurde beschrieben.

4.1 Hauptprogramm

Das Hauptprogramm ist eine in Python geschriebene Flask-Applikation [Fla]. Sie agiert wie ein Server und kann dementsprechend über HTTP-Aufrufe erreicht und angesprochen werden.

Da der Mikroservice alle Suchoperationen in D-BAS übernehmen soll, wurden mehrere Routen definiert, über die verschiedene Suchoperationen ausgeführt werden können. Im Folgenden werden die diversen Suchmöglichkeiten und die damit verbundenen URLs dargestellt. Wie die erstellten Suchanfragen aussehen und wie der Elasticsearch-Server diese verarbeitet wird im darauffolgenden Abschnitt erläutert.

4.1.1 Suchmöglichkeiten und Schnittstellen

In allen Aussagen suchen URL: /all_stmts?text=[suchtext]

D-BAS bietet die Möglichkeit, alle Aussagen zu durchsuchen, um zum Beispiel an eine bestimmte Stelle der Diskussion zu springen (vgl. Abb. 4.1).

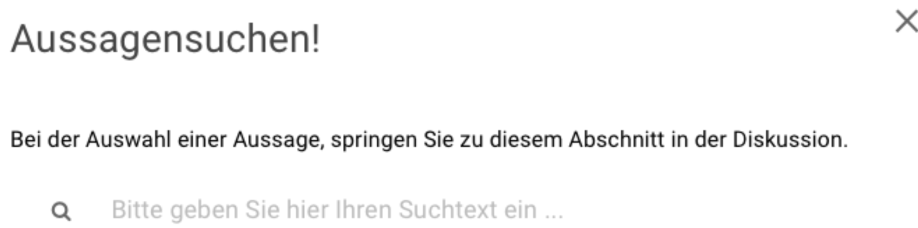

Abbildung 4.1: Screenshot von D-BAS bei Suche in allen Aussagen

Da es hier keinen nutzenbringenden Kontext gibt, werden die Ergebnisse nur anhand der Eingabe ermittelt. Dazu wird im Paramater *text* die Nutzereingabe an den Mikroservice übergeben. Dieser baut daraus die Suchanfrage und stellt sie an den Elasticsearch-Server. Dabei werden **alle** Statements durchsucht.

In allen Textversionen eines Statements suchen URL: /edit_stmt?text[suchtext]&stmt=[statement_uid]

Aktive Nutzer können in D-BAS moderative Aktionen durchführen. Dazu gehört unter anderem das Editieren eines Statements, um sprachliche Fehler auszubessern. Dabei gibt es ebenfalls eine autovervollständigende Eingabemöglichkeit. (vgl. Abb. 4.2).

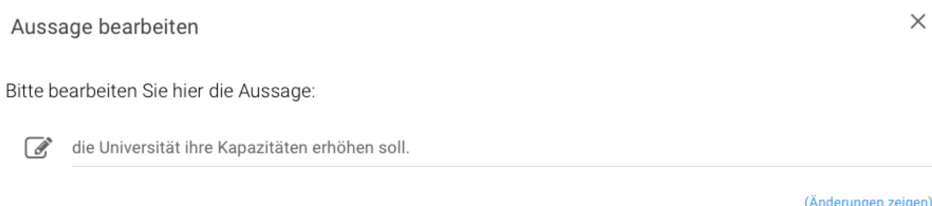

Abbildung 4.2: Screenshot von D-BAS bei Suche in allen Aussagen

An dieser Stelle geht es lediglich um die Korrektur von Rechtschreibung und Syntax, daher werden

nicht alle Statements durchsucht, sondern nur die Textversionen des übergebenen Statements. Das sind alle Varianten, in denen das Statement in D-BAS vorhanden ist.

Start-Positionen suchen URL: /position?text=[suchtext]&issue=[issue_uid]

Zu Beginn einer Diskussion werden dem Nutzer bereits vorhandene Diskussionseinstiege vorgeschlagen. Er hat außerdem die Möglichkeit eine eigene Position einzugeben (vgl. Abb. 4.3).

Ich möchte darüber reden, dass ...
es mehr Seminare geben sollte, die auch benotet sein könnten.
einheitliche Bewertungskriterien für Seminarausarbeitungen und Seminarvorträge eingeführt werden.
mathematische Skripte auch Beweise enthalten sollten.
Nichts von all dem. Ich habe eine andere Idee!
Ich möchte darüber reden, dass
Geben Sie mindestens 10 Zeichen ein
AUSSAGE SPEICHERN!

Abbildung 4.3: Screenshot von D-BAS bei Eingabe einer neuen Position

Hierbei werden alle Statements durchsucht, die als Startpunkt einer Diskussion abgespeichert wurden. Außerdem erhalten Aussagen, die direkt zum übergebenen *issue* (gleichnamiger Parameter) gehören, eine höhere Relevanz (*score*).

Begründungen zu Aussage suchen URL: /justify?text=[suchtext]&user=[user_uid]

Nachdem der Nutzer einer Aussage zugestimmt oder eine Aussage abgelehnt hat, fragt D-BAS nach seinem wichtigsten Grund für diese Entscheidung. Ihm werden wieder einige bereits vorhandene Prämissen vorgeschlagen und er kann eine eigene Prämisse hinzufügen (vgl. Abb. 4.4).

Abbildung 4.4: Screenshot von D-BAS bei einfacher Begründung

Zunächst wird aus der aktuellen History zur übergebenen *user_uid* die zu begründende Aussage ermittelt. Dann werden vom Programm alle Statements (*uids*) bestimmt, die bereits als Prämisse zu dieser Aussage angegeben wurden und entsprechend der vorherigen Angabe des Nutzers unterstützend oder ablehnend sind. Die *uids* werden mit dem Suchtext zu einer Suchanfrage zusammengebaut, die dann an den Elasticsearch-Server übergeben wird. Die hier ermittelten Statements erhalten eine höhere Relevanz, es werden aber **alle** Statements durchsucht.

Begründungen zu Argument suchen URL: /justify_relation?text=[suchtext]&user=[user_uid]

D-BAS konfrontiert den Nutzer immer wieder mit Argumenten anderer Nutzer und erfragt seine Meinung dazu. Diese soll ebenfalls begründet werden. Hier hat der Nutzer wieder die Möglichkeit, eine neue Aussage anzugeben (vgl. Abb. 4.5).

Mr. X hat bisher keine Meinung dazu, dass der "Blick über den Tellerrand" vielleicht für einige Studenten interessant und gewünscht ist, aber mitunter zu weit vom Kern der Informatik abdriftet. Aber er nennt einen Grund dagegen, dass Nebenfächer im Allgemeinen nicht verpflichtend sein sollten. Er sagt, dass sie eine wählbare Alternative zu den Pflichtmodulen in den ersten Semestern bieten und damit auch der Orientierung dienen, ob das Fach das richtige ist oder nicht das Nebenfach.

Was denken Sie darüber?

Jetzt.

Was ist Ihr wichtigster Grund gegen die Aussage, dass sie eine wählbare Alternative zu den Pflichtmodulen in den ersten Semestern bieten und damit auch der Orientierung dienen, ob das Fach das richtige ist oder nicht das Nebenfach?
Weil...

Ich lehne ab, dass sie eine wählbare Alternative zu den Pflichtmodulen in den ersten Semestern bieten und damit auch der Orientierung dienen, ob das Fach das richtige ist oder nicht das Nebenfach ...

...weil

Geben Sie mindestens 10 Zeichen ein

AUSSAGE SPEICHERN!

Abbildung 4.5: Screenshot von D-BAS bei Begründung eines Arguments

Ähnlich wie bei der vorherigen Schnittstelle, wird hier mit Hilfe der Historie des übergebenen Nutzers das betreffende Argument bestimmt. Außerdem wird ermittelt, wie der Nutzer auf die Konfrontation reagiert hat und ob er einen Grund für oder gegen das Argument eingeben möchte. Mittels dieser Informationen werden dann die *uids* der Statements bestimmt, die bereits für bzw. gegen das Argument

benutzt wurden. Hier gibt es mehrere Konstellationen, die im Abschnitt der Evaluierung dargestellt werden.

User suchen URL: /nickname?text=[suchtext]

D-BAS bietet Nutzern die Möglichkeit, untereinander Nachrichten auszutauschen. Bei der Eingabe des Empfängers kommt ebenfalls eine Suche mit Auto-Vervollständigung zum Einsatz, die die öffentlichen Nicknamen (*public_nickname*) aller Nutzer durchsucht.

4.1.2 Ansatz mit phonetischen Algorithmen I

Der erste Ansatz eine Auto-Vervollständigung zu entwickeln, die resistenter gegenüber Rechtschreib- und Tippfehlern ist, basierte auf den phonetischen Algorithmen. Schnell wurde dabei aber deutlich, dass diese dafür nicht geeignet sind. Das größte Problem bei der Suche mit phonetischen Algorithmen ist, dass erst sobald ein Wort vollständig eingegeben wurde, ähnlich klingende Alternativen bestimmt werden können. Das führt dazu, dass die Auto-Vervollständigung während dem Tippen unerwartete oder gar keine Ergebnisse zurückliefert. Aus diesem und weiteren Gründen, welche in der Evaluation näher besprochen werden, wurde ein neuer Ansatz entwickelt, welcher im Folgenden vorgestellt wird.

4.2 Elasticsearch-Server

Für die Entwicklung während dieser Arbeit wurde ein einzelner Elasticsearch-Server konfiguriert, auf dem jeweils ein Index für die Statements, die Textversionen sowie für die User eingerichtet wurde. Der Statements-Index ist für alle Suchanfragen verantwortlich, die auf der aktuellsten Textversion der Statements basieren. Somit ist der Textversionen-Index nur für die Editierungsmöglichkeit der Statements nötig. Die Trennung war notwendig, da die Versionierung, wie sie im Datenmodell von D-BAS vorhanden ist, nicht in Elasticsearch abbildbar war.

Um die Suchanfragen so effizient und schnell wie möglich zu gestalten, wurde die Analyse der Texte zum Zeitpunkt der Indizierung ausgewählt. Für die zu durchsuchenden Felder - *textversions.content* bei den Statements, *public_nickname* bei den Usern sowie *content* bei den Textversionen - wurde jeweils ein *mapping* definiert, in dem den Feldern ein selbst erstellter *autocomplete*-Analyzer zugewiesen wird. Repräsentativ ist in Listing 4.1 das *mapping* für den Statements-Index zu sehen. Für die anderen Indizes ist dieses identisch, bis auf die Feldbezeichnung in Zeile 4.

```
{
    "statement": {
        "properties": {
            "textversions.content": {
                "type": "text",
                "analyzer": "autocomplete",
                "search_analyzer": "standard"
            }
        }
    }
}
```

Listing 4.1: Mapping für Statements

Der autocomplete-Analyzer analysiert die Felder zum Zeitpunkt der Indizierung. Wird jedoch eine Suchanfrage an diese Felder gestellt, so wird diese durch den *standard*-Analyzer analysiert. Bevor die Gründe dafür erläutert werden, soll vorerst der *autocomplete*-Analyzer näher beschrieben werden.

4.2.1 Autocomplete-Analyzer

Der eigens erstellte Analyzer besteht aus zwei hintereinander gereihten Filtern (Listing 4.2 Zeile 15-18) und dem Standard-Tokenizer (Listing 4.2 Zeile 14). Dieser Tokenizer (basierend auf dem *Unicode Text Segmentation*-Algorithmus [DI12]) zerlegt die Texte in einzelne Wörter anhand von Leerzeichen, Zeilenumbrüchen usw. Anschließend normalisiert der *lowercase*-Filter alle Token in Kleinbuchstaben, damit Groß- und Kleinschreibung für die Suche keinen Unterschied darstellt.

Dann kommt ein *edge_ngram*-Filter zum Einsatz (vgl. Listing 4.2 Zeile 5-9 & 17), der alle Token in Fragmente (*n-grams*) zerlegt. Für die Funktionalität der Auto-Vervollständigung ist dabei der Beginn jedes Fragments am Wortanfang verankert. Es werden n-Gramme der Länge 1 bis maximal 15 gebildet.

```
{
"settings": {
   "analysis": {
      "filter": {
         "autocomplete_filter": {
            "type": "edge_ngram",
            "min_gram": 1,
            "max_gram": 15
         }
      },
      "analyzer": {
         "autocomplete": {
            "type": "custom",
            "tokenizer": "standard",
            "filter": [
               "lowercase",
               "autocomplete_filter"
            ]
         }
      }
   }
}
```

Listing 4.2: Erstellen des Analyzers

Beispiel 4.2.1.1 Zerlegung eines Textes durch den autocomplete-Analyzer
Sei der eingegebene Text 'Informatik Vorlesung'. So entstehen zuerst die beiden Token 'Informatik' und 'Vorlesung', welche anschließend normalisiert werden zu 'informatik' und 'vorlesung'. Nun erstellt der egde_ngram-Filter die folgenden n-Gramme:

Token	n-Gramme
informatik	i, in, inf, info, infor, inform, informa, informat, informati, informatik
vorlesung	v, vo, vor, vorl, vorle, vorles, vorlesu, vorlesun, vorlesung

Tabelle 4.1: Beispiel edge_ngram-Filter

Dieser Analyzer wird nur während der Indizierung der Dokumente eingesetzt. Würde man ihn außerdem während der Suchzeit einsetzen, so würde jedes Suchwort wie oben beschrieben zerlegt und für jedes n-Gram die Suche ausgeführt. Das führt zu vielen unerwarteten und ungewollten Treffern.

Stattdessen wird also der *standard*-Analyzer eingesetzt, der den gesuchten Text mit dem bereits bekannten Standard-Tokenizer zerlegt und mit dem lowercase-Filter normalisiert.

4.2.2 Suchanfragen

Da die Analyse bereits im *mapping* hinterlegt ist, sind die Suchanfragen überschaubar. Sie werden über eine RESTful-Schnittstelle an den Elasticsearch-Server gestellt. Alle Parameter werden im Request-Body im JSON-Format übergeben.

Einfache Suchanfrage

Eine Suchanfrage in den Statements für den Text *Informatik Vorlesung* sieht zum Beispiel so aus:

```
{
   "size": 10,
   "_source": ["textversions.uid", "textversions.content"],
   "highlight": {
      "fields": {
         "textversions.content": {
            "number_of_fragments": 0
         }
      }
   },
   "query": {
      "match": {
         "textversions.content": {
            "query": "Informatik Vorlesung",
            "fuzziness": "AUTO"
         }
      }
   }
}
```

Listing 4.3: Beispiel einer einfachen Suchanfrage

Neben des notwendigen *query*-Parameters enthält die Anfrage die optionalen Parameter *size, _source* und *highlight*. Der Paramter *size* gibt an, wie viele Dokumente das Ergebnis maximal beinhalten soll (hier: 10). Über den Parameter *_source* wird hier mitgeteilt, dass lediglich die Felder *textversions.uid* und *textversions.content* im Ergebnis ausgegeben werden sollen. Der Parameter *highlight* sorgt dafür, dass im Inhalt des Feldes *textversions.content* die treffenden Wörter mit <**em**> - Tags (HTML-Tag, das den eingeschlossenen Text kursiv darstellt) maskiert werden.

Die eigentliche Suchoperation wird über den *query*-Parameter definiert. Dazu wird eine *match-query* konstruiert. Diese analysiert den Text (hier mit dem oben beschriebenen Standardanalyzer) und erstellt eine bool'sche *should* Suchanfrage aus den entstandenen Token. Damit ein Dokument als Treffer zählt muss mindestens eine dieser should-Klauseln zutreffen. In der *match-query* wird zuerst das zu durchsuchende Feld als Schlüssel angegeben (vgl. Listing 4.3 Zeile 13). Der zugehörige Wert ist wiederrum ein Objekt mit den Schlüsseln *query*, welches den Suchtext als Wert trägt, und *fuzziness*. Letzterer soll eine Toleranz für Rechtschreibfehler ermöglichen und basiert auf der Levensthein-Distanz. Es kann eine feste Anzahl an Änderungsoperationen oder wie hier, mit dem Parameter „AUTO“, eine von der Wortlänge abhängige Anzahl dieser definiert werden (vgl. Tabelle 4.2).

Wortlänge	Anzahl erlaubte Änderungen
0...2	0
3...5	1
5 +	2

Tabelle 4.2: Anzahl erlaubter Änderungsoperationen nach Wortlänge

Solche Suchanfragen werden dann erstellt, wenn in allen Statements gesucht werden soll und Dokumente keine weiteren Vorraussetzungen erfüllen sollen.

Erweiterte Suchanfrage

In den Szenarien, in denen bestimmte Statements bzw. Statements mit bestimmten Eigenschaften höher bewertet werden sollen, wird eine komplexere Suchanfrage gebaut. Dazu gehört zum Beispiel die Suche nach Start-Positionen. Das nachfolgende Listing 4.4 zeigt ausschnittsweise die Suchanfrage in dieser Situation. Aus Gründen der Übersichtlichkeit werden die statischen Teile (*highlight, size* und *_source*) in den folgenden Listings weggelassen.

```
{
   "query": {
      "bool": {
         "must": {
            "match": {
               "textversions.content": {
                  "query": "informatik",
                  "fuzziness": "AUTO"
               }
            }
         },
         "should": {
            "term": {
               "issue_uid": {
                  "value": "1",
                  "boost": 5
               }
            }
         },
         "filter": {
            "term": {
               "is_startpoint": True
            }
         }
      }
   }
}
```

Listing 4.4: Beispiel einer Suchanfrage für Start-Positionen

Als Erweiterung zur obigen Suchanfrage wurde hier unter anderem die *match-query* unter dem Schlüsselwort *must* in eine *bool-query* gepackt. Dadurch können weitere bool'sche Klauseln hinzugefügt werden. Das ist hier zum einen eine *should*-Klausel (vgl. Listing 4.4 Zeile 12-19), die bewirkt, dass

Statements zu dem angebenen *issue* (hier: 1) einen höheren Score erhalten (*boost*). Zum anderen ist es eine *filter*-Klausel (vgl. Listing 4.4 Zeile 20-24), die dazu führt, dass ausschließlich Statements, die als Startpunkt einer Diskussion gespeichert wurden, für die Suche berücksichtigt werden.

Die zweite erweiterte Suchanfrage, die zum Einsatz kommt ist ausschnittsweise in Listing 4.5 zu sehen.

```
{
   "query": {
      "bool": {
         "must": {
            "match": {
               "textversions.content": {
                  "query": "Informatik",
                  "fuzziness": "AUTO"
               }
            }
         },
         "should": {
            "ids": {
               "values": [2, 3, 77],
               "boost": 5
            }
         }
      }
   }
}
```

Listing 4.5: Beispiel einer Suchanfrage mit gegebenen IDs

Diese Art der Suchanfrage wird dann benutzt, wenn vorher vom Hauptprogramm die IDs von Statements ermittelt wurden, die in bestimmten Konstellationen zu der aktuellen Situation vorkommen. Also zum Beispiel dann, wenn ein Nutzer eine neue Prämisse eingeben möchte. Sie unterscheidet sich nur im Inneren der *bool-query* von der ersten erweiterten Suchanfrage. Hier entfällt die *filter-query* und in der *should*-Klausel befindet sich nun eine *ids-query* (vgl. Listing 4.5 Zeile 12-17). Es wird zum Schlüssel *values* eine Liste von IDs übergeben, sodass Dokumente mit diesen IDs einen höheren Score erhalten und damit eine höhere Relevanz für das Suchergebnis haben.

Kapitel 5

Evaluation

Das Ziel dieser Arbeit war es, die bestehende Auto-Vervollständigung in D-BAS zu verbessern. Das Ergebnis sollte zum einen kontextsensitiv und zum anderen resistenter gegen Rechtschreib- und Tippfehler sein. In diesem Kapitel wird erörtert in wie weit die finale Lösung durch den Elasticsearch-Server diese Anforderungen erfüllt, wieso es eine Verbesserung zur aktuellen Umsetzung ist und an welchen Stellen noch Verbesserungen bzw. Erweiterungen wünschenswert wären. Dazu wird erstmal die aktuelle Umsetzung in D-BAS beschrieben.

Anmerkung: Der Datensatz, der dieser Evaluation zu Grunde liegt, stammt aus einem Feldversuch von D-BAS, der im Mai 2017 stattfand. Darin sind 292 Statements enthalten, die zu insgesamt 260 Argumenten zusammengesetzt wurden.

5.1 Bisherige Vorgehensweise

In der jetztigen Implementation von D-BAS ist für jede der anfangs genannten Eingabemöglichkeiten eine entsprechende Funktion vorhanden, die Statements für die Auto-Vervollständigung aussucht. Im Allgemeinen ist der Ablauf aktuell so, dass eine Menge Statements aus der Datenbank geladen wird und anschließend geprüft wird, ob der vom Nutzer eingegebene Text darin vorkommt.

Die Kriterien, anhand dessen die Statements in den einzelnen Situationen ausgesucht werden, sind in der nachfolgenden Tabelle (Tabelle 5.1) aufgelistet.

Situation	Kriterien
Neue Start-Position	Startpunkt; zu aktuellem Issue zugeordnet
Neue Start-Prämisse	kein Startpunkt; zu aktuellem Issue zugeordnet
Textversion bearbeiten	alle Textversionen des Statements
Alle Statements durchsuchen	zu aktuellem Issue zugeordnet
Neuen Grund	zu aktuellem Issue zugeordnet

Tabelle 5.1: Kriterien für aktuelle Suchfunktion in D-BAS

Nun prüft D-BAS, ob die Nutzereingabe in den gesuchten Statements vorhanden ist. Dabei werden Suchtext und Statementtext zwar in Kleinbuchstaben normiert, jedoch muss in der jetztigen Implementation die Eingabe genauso, wie sie getippt wurde, in den Statements vorhanden sein. Erst nachdem ein Statement mit dem Text gefunden wurde, wird mit Hilfe der Levensthein-Distanz versucht einen eventuellen Unterschied der beiden Text zu finden bzw. zu tolerieren. Da hier die Reihenfolge der Prüfungen falsch ist, werden zu diesem Zeitpunkt keine Rechtschreib- bzw. Tippfehler berücksichtigt. Werden außerdem mehrere Wort eingegeben, so müssen diese in der selben Reihenfolge direkt hintereinander in den Statements vorkommen, um erkannt zu werden.

5.2 Ansatz mit phonetischen Algorithmen II

Mit dem entwickelten phonetischen Ansatz konnten bereits einige Verbesserungen zur aktuellen Umsetzung erreicht werden, jedoch mit neuen Schwachstellen zur Folge. Es konnte die Kontextsensitivität (Vorauswahl der logisch passenden Statements) im selben Maß wie in der finalen Lösungen erreicht werden. Da diese jedoch abgekoppelt von dem eigentlichen Suchverfahren realisiert wurde, ist das weder ein Vor- noch ein Nachteil des phonetischen Ansatzes.

Der größte Vorteil gegenüber dem Bestehenden war die größere Toleranz gegenüber Rechtschreibfehlern und ähnlichen Schreibweisen. Solang die Eingaben entsprechend den dargestellten phonetischen Algorithmen die gleichen phonetischen Codes erzeugt haben, konnten sie beliebig viele Tippfehler beinhalten. So hat zum Beispiel das Wort *Ymvornatyk* nach der Kölner Phonetik immer noch zu Übereinstimmungen mit dem richtigen Text *Informatik* geführt. Andereseits reicht ein Tippfehler, der den Wortklang signifikant ändern würde aus und der Algorithmus findet keine Ergebnisse mehr. Am Beispiel des Wortes Informatik wäre das zum Beispiel *Informatic*. Nach der Kölner Phonetik würde diese beiden Schreibweisen einen unterschiedlichen Code erzeugen und nicht als ähnlich eingestuft.

Neben diesem und dem in Teil 1 beschriebenen Nachteil, ist die Geschwindigkeit dieses Algorithmus nicht überzeugend. Ein Vergleich dessen folgt in einem weiteren Abschnitt dieses Kapitels.

5.3 Bewertung der Kontextsensitivität

Wie bereits in den Grundlagen dieser Arbeit angedeutet, bedeutet Kontextsensitivität für diese Arbeit, dass logisch passende Aussagen für das Ergebnis der Auto-Vervollständigung eine höhere Relevanz haben, als solche die nicht unmittelbar passen würden. Dabei wird jedoch keine Textanalyse vorgenommen und nicht geprüft, ob die Satzbausteine syntaktisch und inhaltlich sinnvoll zusammen passen. Stattdessen wird die Zusammengehörigkeit daraus geschlossen, dass andere Nutzer die Aussagen in dieser Kombination bereits genutzt haben.

Eine weitere Auswertung des vorherigen Nutzerverhaltens, wie es zu Beginn der Bearbeitung angestrebt wurde, ist für die Auto-Vervollständigung ohne eine gute Textanalyse nutzlos. Befindet der Nutzer ein Argument als falsch und möchte widersprechen, so ist es weitesgehend irrelevant, ob er vorher der Start-Position zugestimmt hat oder nicht. Natürlich könnten die Ergebnisse noch präziser ermittelt werden, wenn man die vorherigen Entscheidungen und benutzten Aussagen inhaltlich analysieren und diese dann miteinander in Verbindung bringen könnte.

Um also die korrekte Vorauswahl der Statements in dem finalen Ansatz zu beweisen, wird zuerst formal die Vorgehensweise verifiziert und im Anschluss mit einem Beispiel dargestellt. Dazu wird im folgenden erklärt, wie ein Argument in D-BAS abgespeichert wird und wie die Suchmöglichkeiten aus Sicht von D-BAS aussehen.

5.3.1 Speicherung eines Argumentes in D-BAS

Es gibt in D-BAS zwei Arten von Argumenten. Zum einen sind das Argumente, in denen die Prämisse eine Konklusion begründet und zum anderen Argumente, in denen die Prämisse ein anderes Argument begründet. Um diese Unterscheidung zu vollziehen, gibt es in der Datenstruktur eines Argumentes die Felder *conclusion_uid* sowie *argument_uid*. Während erstere auf ein Statement verweist, verweist letztere auf ein anderes Argument. Da es mehr als eine Prämisse zu jedem Argument geben kann, wird die Verbindung eines Argumentes mit der/ den Prämisse/n über eine *premisegroup* hergestellt. Daher wird in jedem Argument die *premisesgroup_uid* abgespeichert. Die *premisesgroup_uid* wird außerdem in den einzelnen Prämissen abgespeichert. Ob eine Prämisse eine Konklusion in dem Argument unterstützt, wird über das Feld *is_supportive* gespeichert.

5.3.2 Suchmöglichkeiten aus Sicht von D-BAS

Die Suchmöglichkeiten wurden im vorherigen Kapitel bereits aus Sicht des Mikroservices beschrieben und erklärt. Damit die Auswahl der Statements nachvollzogen werden kann, ist es notwendig, die Situationen auch aus Sicht von D-BAS zu betrachten. Bei zwei der Eingabemöglichkeit neuer Aussagen wird eine solche Vorauswahl durchgeführt.

Erster kontextsensitiver Fall Der einfachste Fall, in dem für die Suche der Kontext herangezogen wird, ist bei der Eingabe einer neuen Prämisse zu einer Aussage. Hier ist der aktuelle Schritt in der History des Nutzers wie folgt aufgebaut:

justify/[statement_uid]/{t,f}

Die *uid* in der Mitte gehört dabei zu der zu begründenden Aussage. Der Buchstabe am Schluss gibt an, ob der Nutzer der Aussage zustimmt (*t*) oder sie ablehnt (*f*). Der Mikroservice ermittelt jetzt alle Prämissen der Argumente, die als Konklusion das Statement aus der History und den selben Standpunkt wie der Nutzer haben (zustimmend, ablehnend).

Beispiel 5.3.2.1 Aktueller Schritt von Nutzer X sei *justify/1/t*
Es wird also nach Gründen gesucht, die die Aussage, dass „*eine Zulassungsbeschränkung eingeführt werden soll*“ stützen. Händisch ermittelt wurden die Argumente [1, 2, 3, 4, 38, 86, 97, 98, 175, 236], die Statement 1 als Konklusion haben. Davon sind die Argumente [1, 2, 98] unterstützend. Die dazugehörigen *premisesgroups* sind [1, 2, 96]. Es werden die Statements [2, 3, 77] als Prämissen in den genannten *premisesgroups* ermittelt. Die dazugehörigen Satzbausteine sind:

2 „die Nachfrage nach dem Fach zu groß ist, sodass eine Beschränkung eingeführt werden muss.“

3 „viele Studierenden sich einschreiben, ohne die notwendigen Kompetenzen zu besitzen.“

77 „dadurch die Kurse nicht zum Aussortieren der Studenten genutzt werden“

Es wird erwartet, dass eine Suche nach zum Beispiel dem Wortteil *Stud*, unter anderem die Statements 3 und 77 findet und diese weit oben in den Ergebnissen stehen. Es wird also die URL

/justify?text=Stud&user=X

aufgerufen.

Der Mikroservice erstellt daraufhin die folgende Suchanfrage:

```
{
    "query": {
        "bool": {
            "must": {
                "match": {
                    "textversions.content": {
                        "query": "stud",
                        "fuzziness": "AUTO"
                    }
                }
            },
            "should": {
                "ids": {
                    "values": [2, 3, 77],
                    "boost": 5
                }
            }
        }
    }
}
```

Listing 5.6: Ausschnitt Suchanfrage im ersten Fall

In Zeile 14 des Listing 5.6 sieht man, dass die drei zu priorisierenden Statements richtig ermittelt wurden. Der Auschnitt des Ergebnisses (Listing 5.7) zeigt, dass die beiden Dokumente mit der höchsten Relevanz, den vorher ermittelten entsprechen.

```
{
    "hits": [
        {
            "_id": "77",
            "_score": 8.750988,
            "_source": {
                "textversions": [
                    {
                        "content": "dadurch die Kurse nicht zum Aussortieren
    ↪ der Studenten genutzt werden",
                        "uid": 141
                    }
```

```
12                  ]
13              }
14          },
15          {
16              "_id": "3",
17              "_score": 8.302385,
18              "_source": {
19                  "textversions": [
20                      {
21                          "content": "viele Studierenden sich einschreiben, ohne
   ↪  die notwendigen Kompetenzen zu besitzen.",
22                          "uid": 3
23                      }
24                  ]
25              }
26          },
27          [weitere Ergebnisse]
28      ]
29  }
```

Listing 5.7: Ausschnitt aus Ergebnis im ersten Fall

Zweiter kontextsensitiver Fall Der zweite Fall, in dem der Kontext zur Suche herangezogen wird, tritt dann ein, wenn der Nutzer einen neuen Grund für ein Argument eingeben möchte. Entscheidend für die Auswahl der Statements ist hier, wie der Nutzer auf das Argument, mit dem er konfrontiert wurde, reagiert hat. Dies ist ablesbar am aktuellen Schritt in der History des Nutzers.

Er hat folgenden Aufbau:

justify/[argument_uid]/{t,f}/{undercut,undermine}

Das Schlüsselwort im letzten Teil des Schrittes gibt an, welche der Reaktionsmöglichkeiten auf das Argument der Nutzer gewählt hat.

undermine

Lehnt der Nutzer das Argument ab und möchte widersprechen, so hat der aktuelle Schritt das Keyword *undermine*. In diesem Falle gilt es, Gründe gegen die Prämisse des Arguments (kurz: *a)* zu finden. Es müssen also Statements priorisiert werden, die in Argumenten als Prämisse vorkommen, in

denen *a* als Konklusion dient. Da der Nutzer das Argument abgelehnt hat, sollten die Prämissen nicht unterstützend sein.

Beispiel 5.3.2.2 Aktueller Schritt des Users X sei *justify/248/f/undermine*
Es wird nach Gründen gesucht, wieso die Aussage, dass „*der „Blick über den Tellerrand“vielleicht für einige Studenten interessant und gewünscht ist, aber mitunter zu weit vom Kern der Informatik abdriftet*“ (Prämisse des Arguments, *uid 253*) nicht richtig ist. Es werden jetzt alle Argumente ermittelt, bei denen Statement 253 als Konklusion dient und die Prämissen nicht unterstützend sind. Das ist in diesem Fall das Argument 285 mit der Prämisse 289, zu der der Satzbaustein „*die Anreize aus weiteren Fächern für die Denkweise sehr hilfreich sein können*“ gehört. Diese Aussage sollte im Suchergebnis priorisiert werden. Es wird nun also die URL

/justify_reaction?text=denk&user=X

aufgerufen, die in der oben beschriebenen Situation die Suche nach dem Teilwort *denk* auslöst.

```
{
   "query": {
      "bool": {
         "must":{
            "match": {
               "textversions.content": {
                  "query": "denk",
                  "fuzziness": "AUTO"
               }
            }
         },
         "should": {
            "ids": {
               "values": [289],
               "boost": 5
            }
         }
      }
   }
}
```

Listing 5.8: Ausschnitt Suchanfrage im zweiten Fall 1

Wie in Listing 5.8 Zeile 14 zu sehen, wurde das Statement vom Programm korrekt ermittelt und

bekommt für das Ergebnis eine entsprechend höhere Relevanz. Listing 5.9 zeigt einen Ausschnitt des Suchergebnisses.

```
{
   "hits": [
      {
         "_id": "289",
         "_score": 10.690384,
         "_source": {
            "textversions": [
               {
                  "content": "die Anreize aus weiteren Fächern für die
   ↪ Denkweise sehr hilfreich sein können",
                  "uid": 308
               }
            ]
         }
      },
      [weitere Ergebnisse]
   ]
}
```

Listing 5.9: Ausschnitt aus Ergebnis im zweiten Fall 1

undercut

Alternativ kann der Nutzer die Aussage des Arguments akzeptieren, aber sagen, dass diese die Behauptung nicht stützt. Dann steht im aktuellen Schritt des Nutzers das Keyword *undercut*. Die Vorschlagsliste sollte jetzt bevorzugt die Stamenents beinhalten, die als Prämisse in allen Argumenten vorkommen, die das Argument aus dem Schritt angreifen (also im Feld *argument_uid* dieses Argument haben).

Beispiel 5.3.2.3 Aktueller Schritt von Nutzer X sei *justify/198/t/undercut*
Das Argument 198 sagt folgendes aus: „*Mathematische Skripte sollten keine Beweise enthalten* [conclusion, uid 194]*, da dann die Studierenden keinen Anreiz mehr haben, die Vorlesungen zu besuchen* [premise, uid 196]“. Das einzige Argument, das im aktuellen Datensatz das Argument 198 angreift, ist das Argument 240 mit der Prämissengruppe 243. Dazu gehört die Prämisse 244 mit dem Satzbaustein „*auch andere Vorlesungen, bei denen der Großteil der Vorlesung in Folien enthalten ist, besucht werden*“. Diese sollte nun im Ergebnis eine höhere Relevanz erhalten.

Er findet nun, dass die Prämisse kein guter Grund gegen die Konklusion ist und möchte einen Grund dafür angeben. Er gibt als Stichwort *Vorlesung* ein. Dabei wird der Mikroservice über die URL

/justify_reaction?text=Vorlesung&user=X

aufgerufen und die folgende Suchanfrage gebaut:

```
{
   "query": {
      "bool": {
         "must": {
            "match": {
               "textversions.content": {
                  "query": "Vorlesung",
                  "fuzziness": "AUTO"
               }
            }
         },
         "should": {
            "ids": {
               "values": [244],
               "boost": 5
            }
         }
      }
   }
}
```

Listing 5.10: Ausschnitt aus Suchanfrage im zweiten Fall 2

Die Suchanfrage in Listing 5.10 zeigt, dass das Statement korrekt ermittelt wurde. Das ist ebenfalls im Ergebnis (vgl. Listing 5.11) zu sehen, denn das o.g. Statement hat den höchsten Score erhalten.

```
{
   "hits": [
      {
         "_id": "244",
         "_score": 12.967637,
         "_source": {
            "textversions": [
               {
                  "content": "auch andere Vorlesungen, bei denen der
 ↪  Großteil der Vorlesung in Folien enthalten ist, besucht werden",
                  "uid": 261
               }
            ]
         }
      },
      [weitere Ergebnisse]
   ]
}
```

Listing 5.11: Ausschnitt aus Ergebnis im zweiten Fall 2

5.4 Vergleich Elasticsearch vs. Phonetik vs. bisherigem Ansatz

Nachdem nun gezeigt wurde, dass die Priorisierung von bestimmten Statements korrekt funktioniert, soll in diesem Abschnitt nun verglichen werden, wie *gut* die Ergebnisse der einzelnen Ansätze sind und wie effizient diese arbeiten.

5.4.1 Performance

Neben der Qualität der Ergebnisse, ist die Geschwindigkeit, mit der diese erscheinen eine der wichtigsten Eigenschaften einer Auto-Vervollständigung. Sobald der Nutzer tippt, sollten ihm Ergebnisse präsentiert werden und er sollte nicht auf diese *warten* müssen. Die durchschnittliche Tippgeschwindikeit eines erfahrenen Nutzers liegt bei ungefähr 250 Anschlägen/min. Das entspricht etwa 4 Anschlägen pro Sekunde, sodass ein gutes Verfahren maximal eine Viertel Sekunde benötigt, um die Suchoperation vollständig auszuführen.

In dem nachfolgenden Graphen ist dargestellt, wie schnell die einzelnen Ansätze die Eingabe eines

Wortes je nach Länge bearbeiten[1]. Für die Phonetik wird stellvertretend die *Kölner Phonetik* evaluiert, da die beiden phonetischen Ansätze eine vergleichbare Laufzeit aufweisen.

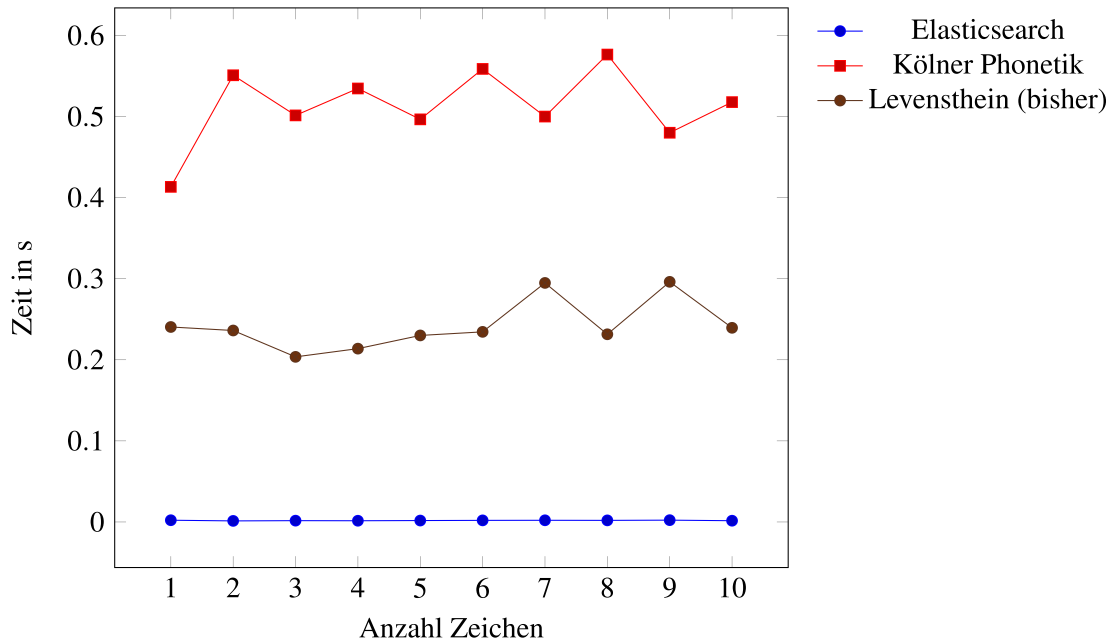

Abbildung 5.1: Vergleich Suchzeit der Algorithmen nach Wortlänge

Wie deutlich zu erkennen ist, benötigt der phonetische Ansatz mit Abstand die meiste Zeit, gefolgt von der bisherigen Implementation (hier kommen ggf. noch einige Hunderstel hinzu, da das highlighten der Ergebnisse hier nicht berücksichtigt wurde). Am schnellsten arbeitet die finale Lösung mit dem Elasticsearch-Server, da hier die Analyse der Texte bereits zum Zeitpunkt der Indizierung durchgeführt wurde. Die anderen beiden Ansätze ermitteln erst zur Suchzeit die phonetischen Codes bzw. die Levensthein-Distanz der gespeicherten Texte.

[1] Die Messungen wurden mit einem i7 Dual Core - Prozessor mit 1.7GHz und 8GB DDR3 RAM ausgeführt. Zu beachten ist außerdem, dass der ElasticSearch Server nur als virutelle Maschine auf dem selben Gerät betrieben wurde.

Sobald der Nutzer mehr als ein Wort eingibt, wird der Unterschied noch deutlicher (vgl. Abb. 5.2).

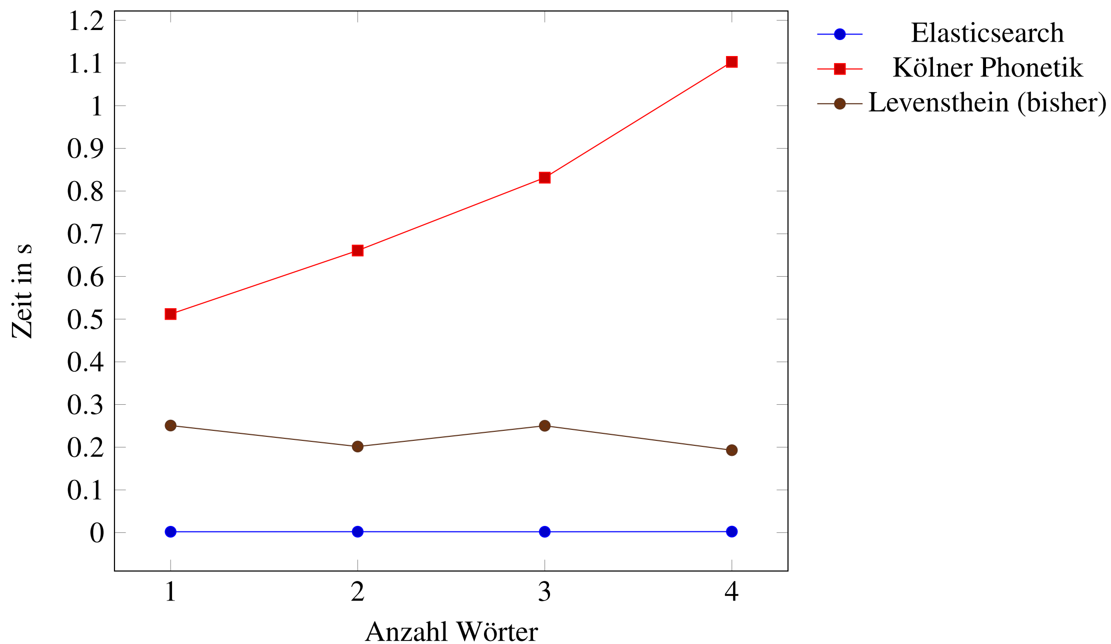

Abbildung 5.2: Vergleich Suchzeit der Algorithmen nach Anzahl Wörtern

Die Zeit, die der phonetische Ansatz benötigt, steigt deutlich an, wenn der Nutzer mehrere Wörter eingibt. Die Elasticsearch-Lösung hingegen benötigt für vier Wörter genauso viel Zeit wie für ein Wort. Die bisherige Umsetzung ist hier im Graphen zwar angegeben, ist aber nur begrenzt konkurrenzfähig, denn, wie bereits angesprochen, findet dieser Ansatz nur dann Ergebnisse, wenn die eingebenen Wörter direkt hintereinander in den gespeicherten Texten vorkommen. Außerdem müssen bei der bisherigen Umsetzung alle Wörter enthalten sein, während bei beiden neu entwickelten Ansätzen auch Ergebnisse ermittelt werden, in denen nur manche der Wörter vorkommen.

Fazit Performance

Die phonetischen Algorithmen sind in der gewählten Umsetzung nicht gut für eine Auto-Vervollständigung geeignet. Das größte Defizit ist die Berechnung der phonetischen Codes zum Suchzeitpunkt. Alternativ könnte das Datenmodell in D-BAS erweitert werden. Die phonetischen Codes könnten dann bei der Eingabe neuer Texte ermittelt und abgespeichert werden. Bei der Suche müssten dann nur noch Vergleiche durchgeführt werden und die Laufzeit würde deutlich geringer sein. Die anderen

Nachteile - starke Einschränkung der möglichen Tippfehler, sowie Vervollständigung erst bei vollständigen Wörtern - sind vorest aber nicht zu beheben und wären weiter vorhanden.

Die bisherige Umsetzung mit der Levensthein-Distanz hat eine gute Laufzeit und liegt unter der oben angesprochenen $1/4$ Sekunde. Da dabei jedoch keine Fehlerkorrektur stattfindet und diese nur mit einem ähnlich aufwendigen Verfahren wie bei der Phonetik (Vergleich jedes gespeicherten Wortes mit der Eingabe) realisierbar wäre, ist die Laufzeit im Allgemeinen nicht aussagekräftig.

Bei der finalen Lösung mit Hilfe des Elasticsearch-Servers werden alle Statements bereits beim Indizieren analysiert, wodurch die Suche am effizientesten ausgeführt werden kann. Durchschnittlich benötigt dieser Weg 0.002 Sekunden, um eine Suchoperation auszuführen und ist dabei unabhängig von der Länge der Worte und der Anzahl derer. Die Indizierung der Statements dauerte bei diesem Datensatz mit 292 Statements etwa $1/10$ Sekunde, muss jedoch jedes mal durchgeführt werden, wenn ein neues Statement oder ein neuer Nutzer in D-BAS hinzugefügt wurde.

5.4.2 Qualtität der Ergebnisse

Die Qualität der Ergebnisse kann formal nicht bestimmt werden, da dies eine sehr subjektive Einschätzung ist. Letztendlich dient die Auto-Vervollständigung als Hilfestellung bei der Eingabe von neuen Texten. Ob diese jedoch wirklich hilfreich ist, kann nur der einzelne Nutzer sagen, weshalb es nötig wäre, einen Feldversuch mit der neuen Implementation durchzuführen und danach die Nutzer zu befragen, wie sie die neue Auto-Vervollständigung bewerten würden.

Ein zweiter entscheidener Faktor, der beschreiben kann, wie gut der gewählte Ansatz ist, ist die Menge an Duplikaten. Sollten mit der neuen Auto-Vervollständigung weniger Duplikate eingetragen werden als vorher, so wäre das ein Indiz dafür, dass die neue Lösung für die Nutzer hilfreicher ist, als es die Vorherige war.

5.5 Verbesserungsmöglichkeiten

Um Duplikate effektiv verhindern zu können, ist es nötig, die Sprache weitergehend zu analysieren. Eine höhere Toleranz gegen Rechtschreib- und Tippfehlern ist zwar sehr wichtig, setzt aber voraus, dass Nutzer stets die *gleichen* Worte benutzen. Damit ist ein Ansatz, der allein darauf beruht, vermeintlich eingeschränkt. Durch den Einsatz des speziellen *n-gram*-Analyzer können zumindest manche Wörter mit einem ähnlichen Wortstamm während der Eingabe ermittelt werden.

Eine sinnvolle Erweiterung, die während der Bearbeitung angestrebt wurde, wäre das Erkennen von Synonymen. Elasticsearch bietet die Möglichkeit, entsprechende Filter bzw. Analyzer zu konstruieren

und eine Synonym-Dateidatenbank zu hinterlegen. Es hat sich jedoch gezeigt, dass eine Verknüpfung des Synonym-erkennenden Analyzers mit dem für die *search-as-you-type*-Funktionalität optimierten *autocomplete*-Analyzer nicht sinnvoll möglich war. Es wurden mehrere Lösungen zwischenzeitlich erreicht, deren Nachteile jedoch deutlich überwiegten und somit nicht für die finale Implementierung berücksichtigt wurden. Das zentrale Defizit dieser Lösungen ähnelt dem Verhalten der Phonetik. Nur die Synonyme von komplett ausgeschriebenen Wörter, die keinen Rechtschreibfehler enthalten, konnten für die Suche benutzt werden. Während man ein Wort tippt, erhielt man zudem kaum nachvollziehbare Ergebnisse, da die eingebenen Wortteile durch den Synonym-Analyzer in eine Vielzahl von Worten übersetzt wurde, mit der dann im Index gesucht wurde.

Ein weiterer, allerdings weniger relevanter Nachteil enstand durch das highlighten der Treffer. Dadurch, dass zwei Analyzer auf ein Feld angewandt wurden, wurden ebenfalls zwei unterschiedliche, gehighlightete Felder im Ergebnis ausgegeben. Hier müsste also, bevor die Ergebnisse ausgegeben werden, bestimmt werden, welche Version anzuzeigen ist.

Kapitel 6

Zusammenfassung und Ausblick

6.1 Zusammenfassung

Im Rahmen dieser Arbeit wurde eine Auto-Vervollständigung für die Such- bzw. Eingabemöglichkeiten in D-BAS als Mikroservice entwickelt. Die Aufgabe des Mikroservices ist es, für alle Suchoperationen entsprechend der Eingabe des Nutzers Vorschläge zu ermitteln. In den meisten Szenarien (außer bei der Suche nach einem Nutzer und dem Editieren eines Statements) sollten diese aus einer möglichst großen Menge von Statements ausgewählt werden und dabei eine abhängig vom Kontext bewertete Relevanz für das Ergebnis erhalten. Dafür bietet der entwickelte Mikroservice D-BAS sechs Routen als Schnittstellen an, die jeweils für eine separate Situation optimierte Vorschläge ermitteln.

Zu Beginn wurde die Umsetzung mit Hilfe von phonetischen Algorithmen angestrebt, welche sich jedoch als nicht geeignet erwiesen haben. Das finale Ergebnis benutzt die Open-Source-Software Elasticsearch. Die Kombination des Hauptprogramms, welches für die Kontextauswertung und Steuerung verantwortlich ist, mit dem Elasticsearch-Server, der die Suchanfragen durchführt, ergibt eine sehr effiziente Implementierung einer kontextsensitiven Auto-Vervollständigung.

Um diese Effizienz zu erreichen, werden alle Statements bereits zum Zeitpunkt ihrer Indizierung im Elasticsearch-Server analysiert. Für die Analyse kommt ein Analyzer zum Einsatz, der speziell für die *search-as-you-type*-Funktionalität entwickelt wurde. Dadurch werden zum Suchzeitpunkt lediglich Vergleiche ausgeführt, sodass die durchschnittliche Dauer einer Suchanfrage bei rund 0.002 Sekunden liegt.

Der Kontext, der für die Vorschläge zur Hilfe genommen wird, basiert zum einen auf der aktuellen Entscheidung des Nutzers und zum anderen auf den bisherigen Eingaben anderer Diskussions-Teilnehmer. So werden solche Statements priorisiert, die bereits in dieser Situation in D-BAS vorhanden sind und die außerdem die gleiche Einstellung gegenüber einer Aussage bzw. eines Arguments repräsentieren, wie der Nutzer zuletzt geäußert hat. Um die Vorauswahl durchzuführen, wird der letzte

Schritt des Nutzers in der Diskussion ausgewertet und anschließend die Datenbank von D-BAS nach entsprechenden Statements durchsucht. Dem Elasticsearch-Server werden neben dem eingegebenen Text die ermittelten Statements mitgeteilt, damit dieser diesen bei der Suche eine entsprechend höhere Relevanz zuordnen kann.

6.2 Ausblick

Neben der Weiterentwicklung der Auto-Vervollständigung, wäre es sinnvoll, das Ergebnis dieser Arbeit durch einen Feldtest zu evaluieren. Die Qualität der Ergebnisse ist eine äußerst subjektive Einschätzung der Nutzer. Man sollte herausfinden, ob die Diskutanten die neue Auto-Vervollständigung als hilfreich einschätzen und ob sie die Vorschläge als passend bezeichnen würden. Während einem solchem Feldversuch könnte man ebenfalls auswerten, wie viele Duplikate eingegeben wurden und die Anzahl mit einem vorherigen Feldversuch vergleichen. Mit diesen Informationen kann dann die Qualität der Auto-Vervollständigung bewertet werden und man erkennt ihre Schwachstellen bzw. ihre Stärken.

Im Allgemeinen ist die Weiterentwicklung von Auto-Vervollständigungssystemen stark abhängig von den Fortschritten in der Textanalyse. Wie bereits angesprochen, wäre die Erkennung von Synonymen eine deutliche Verbesserung und würde zu deutlich mehr Ergebnissen führen. Außerdem hilfreich für Sprachen, in denen viele zusammengesetze oder auf dem selben Wortstamm basierende Wörter vorkommen, wie zum Beispiel Deutsch, wäre ein effektiver *Stemming*-Algorithmus. Ein idealer Stemming-Algorithmus erkennt zum Beispiel die Mehrzahl von Wörtern, ihre Konjugation oder ihre geschlechtergerechte Schreibweise und normiert sie auf einen einheitlichen Wortstamm.

Literaturverzeichnis

[DI12] DAVIS, Mark; IANCU, L: Unicode text segmentation. In: *Unicode Standard Annex* 29 (2012).

[Dud] *Dudeneintrag Phonetik*. http://www.duden.de/node/681106/revisions/1362283/view, Abruf: 04.07.2017.

[Ela17] ELASTIC: *Elasticsearch (Version 5.4.0)*. https://www.elastic.co/guide/en/elasticsearch/reference/current/getting-started.html. Version: 2017, Abruf: 04.07.2017.

[Fla] *Flask (A Python Webframework)*. http://flask.pocoo.org, Abruf: 04.07.2017.

[Gen17] GENENGER, Stefan: *Recommender Systems für Webbasierte Argumentationssysteme*, Heinrich-Heine-Universität Düsseldorf, Bachelorarbeit, April 2017

[goo17a] *Automatische Vervollständigung bei der Suche nutzen*. https://support.google.com/websearch/answer/106230?hl=de. Version: 2017, Abruf: 04.07.2017.

[goo17b] *Google - Alles über die Suche*. https://www.google.com/intl/de_ALL/insidesearch/. Version: 2017, Abruf: 04.07.2017.

[goo17c] *Statistic Brain - Google Index Size*. http://www.statisticbrain.com/total-number-of-pages-indexed-by-google/. Version: März 2017, Abruf: 04.07.2017.

[goo17d] *Über Google Instant*. https://www.google.com/intl/de_ALL/insidesearch/features/instant/about.html. Version: 2017, Abruf: 04.07.2017.

[GT15] GORMLEY, Clinton; TONG, Zachary: *Elasticsearch: The Definitive Guide*. 1st. O'Reilly Media, Inc, 2015

[Höl08] HÖLZLE, Urs: *Update to Google Suggest*. https://googleblog.blogspot.de/2008/09/update-to-google-suggest.html. Version: September 2008.

[KBBM16] KRAUTHOFF, Tobias; BETZ, Gregor; BAURMANN, Michael; MAUVE, Martin: Dialog-Based Online Argumentation. In: *Computational Models of Argument*, Potsdam, 2016, S. 33–40.

[Kib17] *Kibana (Version 5.4)*. https://www.elastic.co/guide/en/kibana/current/index.html. Version: 2017, Abruf: 04.07.2017.

[OR18] ODELL, M; RUSSELL, R: The soundex coding system. In: *US Patents* 1261167 (1918).

[Phi90] PHILIPS, Lawrence: Hanging on the metaphone. In: *Computer Language* 7 (1990), Nr. 12 (December).

[Phi00] PHILIPS, Lawrence: *The Double Metaphone Search Algorithm, C/C++ Users Journal, June 2000*. 2000.

[Pos69] POSTEL, Hans J.: Die Kölner Phonetik. Ein Verfahren zur Identifizierung von Personennamen auf der Grundlage der Gestaltanalyse. In: *IBM-Nachrichten* 19 (1969), S. 925–931.

[Wul15] *Wulff gegen Google*. https://www.heise.de/newsticker/meldung/Autocomplete-Funktion-Bettina-Wulff-schliesst-Vergleich-mit-Google-2518426.html. Version: Januar 2015, Abruf: 02.07.2017.

Ehrenwörtliche Erklärung

Hiermit versichere ich, die vorliegende Bachelorarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt zu haben. Alle Stellen, die aus den Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Düsseldorf, 06 Juli 2017 Frederik Grieshaber

Bitte hier

die Hülle mitsamt DVD einkleben

Diese DVD enthält:

- Eine *pdf* Version der Bachelorarbeit
- Alle LaTeXund Grafik Dateien mitsamt dazugehörigen Skripten, die verwendet wurden
- Der Quellcode, der während der Bachelorarbeit erarbeitet wurde
- Alle während der Evaluation angefallenen Messdaten
- Alle referenzierten Webseiten
- Das Image der virtuellen Maschine, auf der Elasticsearch installiert wurde
- Den in der Arbeit verwendeten Datensatz der D-BAS Datenbank