UNIVERSITÄT DUISBURG ESSEN

Offen im Denken

PALUNO
The Ruhr Institute for Software Technology

Institut für Informatik und Wirtschaftsinformatik (ICB)
Lehrstuhl für Software Engineering, insb. mobile Anwendungen
Prof. Dr. Volker Gruhn

Techniken der Computerlinguistik zur Verbesserung von Suchfunktionen in der Software-Entwicklung

Bachelorarbeit

vorgelegt der Fakultät für Wirtschaftswissenschaften
der Universität Duisburg-Essen (Campus Essen) von

Leon Zimmermann
Laddringsweg 8
45219 Essen
Matrikelnummer: 3080384

Essen, den 23. September 2023

Betreuung:	Wilhelm Koop, Sascha Feldmann
Erstgutachter:	Prof. Dr. Volker Gruhn
Zweitgutachter:	Prof. Dr. Klaus Pohl
Studiengang:	Angewandte Informatik - Systems Engineering (B. Sc.)
Semester:	10

Abstract

Zusammenfassung

Softwareentwickler müssen sich bei ihrer Arbeit Informationen zusammensuchen, welche sie für die weitere Arbeit benötigen. So muss ein Softwareentwickler bei der Implementierung eines Features die Intention des Features kennen. Solche Informationen können in Wissensdatenbanken hinterlegt sein. Um dort die gewünschten Informationen zu finden können Suchfunktionen verwendet werden. Insbesondere, wenn dem Softwareentwickler im Vorfeld nicht klar ist, wo er die gewünschten Informationen in der Wissensdatenbank finden kann. Aber nicht immer liefert diese Suchfunktion die gewünschten Informationen.

In dieser Arbeit wird eine semantische Suchfunktion entwickelt. Diese soll eine Verbesserung zu bestehenden Suchfunktionen bieten, wie beispielsweise die Suchfunktion von Confluence. Um festzustellen, ob die neue Suchfunktion *besser* ist als die bestehende, werden Methoden und Kriterien zur Evaluierung von Suchfunktionen erörtert. Anhand dieser Kriterien werden die neu implementierte Suchfunktion und die bestehende Suchfunktion von Confluence in einer Studie verglichen. In der Studie werden beispielhafte Sucheingaben definiert, sowie die erwarteten Ergebnisse. Die Definition der Sucheingaben erfolgt auf Basis von Anwendungsfällen. Die Anwendungsfälle beschreiben Situationen, in denen es realistisch ist, dass ein Softwareentwickler die Suchfunktion einer Wissensdatenbank verwendet. Anhand von Argumentationen werden entsprechend realistische Sucheingaben erstellt.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

ANN Approximate Nearest Neighbor

HNSW Hierarchical Navigable Small Worlds

NLP Natural Language Processing

VSM Vector Space Model

1. Einleitung

Für die tägliche Arbeit benötigt ein Softwareentwickler Informationen, welche über den Code, an dem er arbeitet, hinausgehen. Um an diese Informationen zu kommen kann der Softwareentwickler eine Person suchen, welche ihm die gewünschte Information geben kann. Diese Person kann er im Projekt finden, oder über Websites, wie StackOverflow. Darüber hinaus wird in vielen Projekten eine Wissensdatenbank angelegt, welche Informationen enthält, welche spezifisch für das Projekt sind. Eine solche Wissensdatenbank ist Confluence. Sie bietet eine Suchfunktion, welche es dem Softwareentwickler erleichtern soll, die gewünschten Informationen zu finden. Beispiele für Informationen sind die Spezifikationen oder Dokumentationen eines Teiles der Software, mit welcher der Softwareentwickler gerade arbeitet. Oder aber auch Best-Practices, Guides, oder Informationen darüber, wie die Software gestartet oder ausgeliefert wird. Auch Informationen über den Projektplan sind für einen Softwareentwickler von Bedeutung. Aber nicht immer finden Softwareentwickler die gewünschten Informationen mithilfe der Suchfunktion. Ziel dieser Arbeit soll es sein, mithilfe von Wissen über Suchalgorithmen und Algorithmen aus dem Natural Language Processing zu zeigen, wie sich bestehende Suchfunktionen verbessern lassen.

Software, wie chatGPT[1] zeigt, dass Large Language Models eine valide Möglichkeit zur Information Extraction sind. Es ist denkbar, dass Suchfunktionen für Softwareentwickler durch Large Language Models verbessert werden können. Auch die Verwendung von vorgefertigten semantischen Netzen, welche in Knowledge Bases, wie DBPedia[2] zu finden sind, sind als Lösung für dieses Problem denkbar (Li et al. 2006). Sowohl die Verwendung von semantischen Netzen als auch die Verwendung von Vektordatenbanken kann als semantische Suche verstanden werden. Später soll noch einmal auf den genauen Unterschied zwischen den beiden Methoden eingegangen werden. In dieser Arbeit soll es lediglich um die Verwendung von Vektordatenbanken gehen.

1.1. Vorgehensweise

Die Gründe, warum eine gewünschte Information schwierig zu finden ist, sind vielzählig. Manchmal kennt der Softwareentwickler nicht das genaue Wording, um die gewünschten Informationen zu finden. Manchmal ist das Abstraktionslevel der gefundenen Informationen nicht das, welches sich der Softwareentwickler gewünscht hat. Beispielsweise, wenn eine allgemeine Definition von Domänenobjekten ge-

[1] https://openai.com/blog/chatgpt
[2] https://www.dbpedia.org/

sucht wird, aber eine Spezifikation eines Anwendungsfalls gefunden wird, in welchem das Domänenobjekt lediglich erwähnt wird. Es werden folgende Teilschritte durchlaufen, um die Suchfunktion von Wissensdatenbanken, wie Confluence, zu verbessern:

- **Definition von Anwendungsfällen:** Es werden zuerst Anwendungsfälle definiert. Damit wird das Problem der Qualität einer Suchfunktion heruntergebrochen in Teilprobleme. Die Anwendungsfälle beschreiben die konkreten Situationen, in welchen ein Softwareentwickler eine Suche nutzen könnte. Das hilft später dabei mehrere Suchfunktionen miteinander vergleichen zu können. Denn anhand der Anwendungsfälle können realistische Suchanfragen definiert werden. Diese Suchanfragen können an zwei verschiedene Suchfunktionen übergeben werden. Anschließend können die gefundenen Ergebnisse verglichen werden. Die genaue Vorgehensweise für diesen Vergleich wird in Kapitel 6 und Kapitel 7 erklärt. Außerdem bietet die Aufteilung in Anwendungsfälle bereits Aufschluss über die möglichen Verbesserungen, welche gemacht werden können. Hierauf wird in Kapitel "Erklärung des theoretischen Hintergrunds" und "Implementierung" weiter eingegangen.

- **Erklärung des theoretischen Hintergrunds:** Es wird die Theorie für die Implementierung einer Suchfunktion erläutert. Hier wird erklärt, welche Suchfunktionen es gibt. Außerdem werden Verfahren zur Indizierung von Dokumenten erklärt. Darüber hinaus werden NLP-Techniken erläutert, mit welchen Informationen aus gefundenen Dokumenten extrahiert werden können.

- **Implementierung:** Es wird eine neue Suchfunktion anhand der Informationen des theoretischen Hintergrunds implementiert. Es wird die Weaviate[3] Vektordatenbank verwendet, um Dokumente zu indizieren.

- **Herausarbeitung von Evaluationsmethoden und -Kriterien:** Es werden Methoden für die Bewertung herausgearbeitet. Damit wird die Frage beantwortet, wann eine Suchfunktion *gut* ist. Das hier erläuterte Wissen wird für die Durchführung der Studie benötigt.

- **Durchführung einer Studie:** Um festzustellen, ob die Implementierung eine Verbesserung darstellt, muss eine Studie durchgeführt werden. Aufgrund des Scopes der Arbeit wird nur eine rudimentäre Studie durchgeführt. Die Ergebnisse werden dargestellt und diskutiert. Dabei wird auch darauf eingegangen, an welchen Stellen die Studie weiter ausgearbeitet werden muss, um präzise Ergebnisse liefern zu können. Außerdem wird der Versuchsaufbau beschrieben und die gemessenen Daten. Die Ergebnisse werden interpretiert und es wird ein Schluss gezogen.

[3] https://weaviate.io/

1.2. Verwandte Arbeiten

Es gibt einige Arbeiten, welche die gleichen oder sehr ähnliche Probleme adressieren. Zum einen sind dies Arbeiten, welche verschiedene Arten von Suchfunktionen untersuchen. Andere Arbeiten untersuchen, wie sich die Qualität einer Suchfunktion messen lässt. Nochmals andere Arbeiten untersuchen *Downstream-Tasks* von Suchfunktionen, also Algorithmen, welche das Ergebnis einer Suche verarbeiten, um die gleiche oder eine andere Anforderung an ein System umzusetzen.

So wird in *Automatic Query Reformulations for Text Retrieval in Software Engineering* von Haiduc et. al. ein System zur Verbesserung von Suchanfragen vorgeschlagen. Ausgangspunkt für das Paper ist das Problem der Traceability zwischen Code und anderen Softwareentwicklungs-Artefakten. Traceability bedeutet, dass sich von einer Stelle im Code, auf die entsprechenden Stellen in anderen Artefakten zurückschließen lässt. Ein Anwendungsfall für eine solche Traceability-Funktionalität ist *Feature Location*, also das Finden der Spezifikation eines Features, wenn nur der Code vorhanden ist. Das System, welches von Haiduc et. al. vorgeschlagen wird, verwendet Query Reformulations, um die Traceability herzustellen. Query Reformulation bedeutet, dass das System den Softwareentwickler bei der Eingabe einer Suchanfrage zur Suche nach den passenden Artefakten unterstützt. Dazu gibt der Softwareentwickler zunächst eine Suchanfrage ein, und markiert diejenigen Ergebnisse, welche am relevantesten für ihn sind. Auf Grundlage der gewählten Ergebnisse und mithilfe eines Machine Learning Algorithmus werden nun Vorschläge für eine verbesserte Suchanfrage gemacht. Dabei gibt es verschiedene Strategien. Wenn der Softwareentwickler zu Beginn eine sehr lange Suchanfrage eingegeben hat, dann kann das System eine Reduktion der Suchanfrage vorschlagen. Hat der Softwareentwickler dagegen lediglich einen Suchbegriff angegeben, so kann das System eine Erweiterung der Suchbegriffe vorschlagen. Dazu greift das System auf Synonyme des eingegebenen Suchbegriffes zurück (Haiduc et al. 2013).

In dem Paper *From Word Embeddings To Document Similarities for Improved Information Retrieval in Software Engineering* von Ye et. al. wird beschrieben, wie Word Embeddings dazu verwendet werden können, um Traceability zwischen Code und anderen Softwareentwicklungs-Artefakten herzustellen. Word Embeddings sind eine Datenstruktur, welche einem Wort einen Vektor in einem n-dimensionalen Raum zuweist. Anhand dieses Vektors kann die Ähnlichkeit zwischen Wörtern beschrieben werden. Ähnliche Wörter haben eine geringe Distanz im n-dimensionalen Raum. Unähnliche Wörter haben eine hohe Distanz. Der Algorithmus, welcher die Ähnlichkeit der Wörter bestimmt, macht Gebrauch von der Distributional Hypothesis. Dieser besagt, dass Wörter, welche im gleichen Kontext verwendet werden, eine ähnliche Semantik besitzen. Hermit wird also die Ähnlichkeit der Wörter bestimmt. Dieses Verfahren wird nun sowohl auf den Code angewendet als auch auf die Softwareentwicklungs-Artefakte (Ye et al. 2016).

In dem Paper *Information Retrieval Models for Recovering Traceability Links between Code and Documentation* verwenden Antoniol et. al. einen ähnlichen Ansatz, wie Ye et. al. Auch hier werden Word Embeddings verwendet um Softwareentwicklung Artefakte gegen den Code zu matchen. Hier durchlaufen die Artefakte und der Code zwei verschiedene Pipelines. Die Wörter der Artefakte in natürlicher Sprache werden in lowercase umgewandelt. Anschließend werden Stoppwörter entfernt. Zuletzt werden Flexionen entfernt. Aus dem Code werden zunächst Identifier extrahiert. Identifier, welche mehrere Wörter unter Verwendung von CamelCase oder snake_case beinhalten, werden in die einzelnen Wörter aufgeteilt. Anschließend werden die Identifier auf die gleiche Art und Weise normalisiert, wie die Wörter der Softwareentwicklungs-Artefakte. Dann erfolgt sowohl für die Identifier als auch für die Wörter aus den Artefakten die Indizierung, also die Umwandlung in Word Embeddings (Antoniol et al. 2000).

In dem Paper *TaskNav: Task-based Navigation of Software Documentation* von Treude et. al. geht es um die Entwicklung einer Oberfläche, welche die Suche von *Tasks* ermöglicht. Dabei ist unter Task eine Operation im Code zu verstehen. Das Paper beschreibt einen Task als Verben, welche mit einem direkten Objekt oder einer Präposition in Verbindung stehen. Die Autoren nennen die Phrasen *get iterator* und *get iterator for collection* als Beispiele. Die Software analysiert nun die gesamte Dokumentation und extrahiert Tasks. Die Tasks werden in einen Index geschrieben, sodass der Softwareentwickler nach ihnen suchen kann. So wie die vorherigen Paper soll auch dieses Paper eine Brücke zwischen Dokumentation und Code schaffen (Treude et al. 2015).

Das Paper *Estimating the recall performance of Web search engines* von Clarke und Willet misst die Qualität von Suchfunktionen des World Wide Webs anhand dessen Recalls. Um dies zu ermöglichen wird ein Datensatz generiert, welcher Sucheingaben beinhaltet, sowie alle relevanten Dokumente für eine Sucheingabe (Clarke und Willett 1997). Die gleichen Metriken werden auch in *Methods for measuring search engine performance over time* von Bar-Ilan verwendet (Bar-Ilan 2002). Hier wird ebenfalls auf die Problematik der Messung des Recalls eingegangen. Es wird erläutert, dass zur Messung des Recalls a-priori bestimmt werden muss, welche Dokumente als relevant für eine gegebene Sucheingabe erachtet werden sollten. In dem Paper wird eine Referenz genannt, welche behauptet, dass die Bestimmung der Relevanz lediglich dem Nutzer mit dem Bedürfnis nach der Information überlassen ist. Es wird eine weitere Referenz genannt, welche behauptet, dass die Bestimmung der Relevanz durch ein Experten-Panel durchgeführt werden sollte. *On Search Engine Evaluation Metrics* von Sirotkin betrachtet verschiedene Ansätze zur Messung der Performance von Suchfunktionen. Neben den bereits genannten Metriken von Precision und Recall werden andere Metriken, wie Mean Reciprocal Rank und Maximal Marginal Relevance.

Suchfunktionen sind Document Retrieval Systeme. Sie liefern Dokumente, welche zu der Sucheingabe des Nutzers passen. Document Retrieval Systeme sind eine Unterkategorie von Information Retrieval Systemen. Information Retrieval Syste-

me liefern auf Anfrage Informationen an den Nutzer. Im Fall einer Suchfunktion werden Dokumente geliefert, welche diese Information beinhalten. Mithilfe der Dokumente ist der Ort, an dem sich die, vom Nutzer gewünschte, Information befindet eingegrenzt. Nichtsdestotrotz muss der Nutzer aus dieser Eingrenzung die gewünschte Information manuell extrahieren. Ganz im Gegensatz zu Question Answering Systemen. *A survey for Efficient Open Domain Question Answering* von Zhang et. al. untersucht verschiedene Herangehensweisen zur Implementierung von Open-Domain Question Answering Systemen (Zhang et al. 2023). Darüber hinaus schließt das Paper auf essenzielle Techniken für Open-Domain Question Answering. Open-Domain Question Answering Systeme beantworten allgemeine Fragen eines Nutzers, z.B. basierend auf Informationen von Wikipedia. Closed-Domain Question Answering Systeme beantworten dagegen Fragen im Kontext einer spezifischen Domäne, z.B. basierend auf unternehmensinternen Informationen.

2. Definition von Anwendungsfällen

In diesem Kapitel werden Anwendungsfälle ausgewählt, für welche später Lösungsansätze entwickelt werden. Die Anwendungsfälle beschreiben die Situationen, in denen Softwareentwickler die Suchfunktionen von Wissensdatenbanken verwenden könnten. Zur Identifikation von Anwendungsfällen wurde zunächst Literatur herangezogen. Die Literatur ist bereits unter den verwandten Arbeiten aufgeführt. In den verwandten Arbeiten wurden Arbeiten genannt, welche ähnliche Probleme lösen sollen. Diese fokussieren sich vorallem auf die *Feature Location*, *Bug Localization* und die Traceability zwischen Code und anderen Artefakten. Mit anderen Worten: Die Suchfunktion soll eine Brücke zwischen Code und Dokumentation herstellen. Neben den beiden Anwendungsfällen Feature Location und Bug Localization aus der Literatur, konnten noch weitere Anwendungsfälle ermittelt werden. So ist das Onboarding im Projekt ein Anwendungsfäll für die Verwendung der Suchfunktion einer Wissensdatenbank. Ein weiterer Anwendungsfall ist das Auffinden von Informationen über das Projektmanagement.

Die folgenden Kapitel beschreiben die genannten Anwendungsfälle und erläutern, warum sie als Anwendungsfälle ausgesucht wurden. Bei dem Vergleich zwischen Suchfunktionen werden die Anwendungsfälle herangezogen, um Sucheingaben und erwarteten Dokument zu generieren. Anhand dieser wird die Studie die Precision der Suchfunktionen ermitteln.

2.1. Onboarding im Projekt

Wenn ein neuer Softwareentwickler in einem Softwareprojekt startet, dann muss er sich zunächst einmal mit dem Projekt vertraut machen. Das bedeutet, dass er verstehen muss, was das Projekt eigentlich ist. Er muss verstehen, was das eigentliche Problem des Kunden ist. Außerdem muss er verstehen wie die Software dieses Problem löst.

Für das Onboarding im Projekt muss der Softwareentwickler sehr allgemeine Informationen über das Projekt finden können. Er könnte Dinge suchen, wie einen Projektüberblick oder ein Glossar. Neben diesen allgemeinen Informationen muss sich der neue Softwareentwickler mit dem Code vertraut machen. Er muss verstehen, welche Technologien verwendet werden, welche Best-Practices, Code-Styles, Guidelines, Prozesse und Quality Gates eingehalten werden müssen. Und er muss verstehen, wie die Software lokal oder in einer Testumgebung ausgeführt werden kann.

Die Leitung eines Projektes wünscht sich eine möglichst schnelle Einarbeitung von neuen Mitarbeitern. Dazu können bereits etablierte Mitarbeiter herangezogen werden, welche den neuen Mitarbeiter bei der Einarbeitung unterstützen. Der Nachteil besteht darin, dass hierdurch Kapazitäten gebunden werden, welche für die aktive Entwicklung der Software benötigt werden. Daher kann das Zurückgreifen auf eine Wissensdatenbank durch den neuen Mitarbeiter sinnvoll sein. Dazu kann der neue Mitarbeiter das Inhaltsverzeichnis der Wissensdatenbank verwenden, wenn es vorhanden ist. Dieses hilft dem Mitarbeiter dabei Informationen zu bestimmten Themenbereichen zu finden. Es grenzt die Antwort auf eine Frage auf einen bestimmten Bereich ein, so wie es auch eine Suchfunktion tut. Nichtsdestotrotz bietet eine Suchfunktion die Möglichkeit dieses Inhaltsverzeichnis automatisch auf Basis einer Sucheingabe zu durchlaufen. Damit findet der neue Mitarbeiter schneller die Informationen, die er sucht. Darüber hinaus kann die Suchfunktion spezifischere Ergebnisse liefern. So kann die Suchfunktion dem Nutzer bereits die relevantesten Stellen in den relevantesten Dokumenten liefern. Und die Suchfunktion kann dem Nutzer die Informationen aus diesen relevantesten Stellen zusammenfassen. Voraussetzung hierbei ist eine Suchfunktion, welche gut darin ist die relevantesten Dokumente und die relevantesten Stellen aus diesen Dokumenten zu extrahieren. Im weiteren Verlauf wird erläutert, welche Ansätze es gibt, um diese Voraussetzung zu erfüllen, und wie aus den Dokumenten die relevantesten Stellen mithilfe von Passage Retrieval ermittelt werden können. Außerdem wird erläutert, wie Retrieval Augmented Generation die gefundenen Informationen aufbereitet.

2.2. Implementierung nach Spezifikation

Bei der Implementierung von neuen Anforderungen ist es wichtig, dass sich der Softwareentwickler an die Spezifikation hält. Nur so bekommt der Kunde die Software, die er sich gewünscht hat. Dazu sollte der Softwareentwickler alle relevanten Dokumente finden, die zu der Spezifikation dazugehören. Zuerst sollte er die Feature-Spezifikation selbst finden können. Er sollte die Dokumentation der damit einhergehendenden Prozesse finden, und auch die Domänenobjekte, welche für die Implementierung relevant sind. Er sollte Diagramme finden können, welche zu dem Anwendungsfall gehören, und auch die weiteren Dokumente, welche den Kontext der Anforderung erläutern.

Auch nachdem ein Feature nach Spezifikation umgesetzt wurde, ist es weiterhin wichtig die Spezifikation einfach leicht zu können. Der Abgleich mit einer Spezifikation ist notwendig, um Testfälle zu schreiben, und zu prüfen, ob das Verhalten der Anwendung korrekt ist. Es gibt Fehler, welche erkennbar sind, ohne dafür die Spezifikation heranzuziehen. Eine Anwendung, welche einfriert oder abstürzt ist ein Beispiel für einen solchen Fehler. Dennoch können Fehler auch domänen- und kontextspezifisch sein, wodurch die Spezifikation als Referenz notwendig ist. Die Spezifikation zum Abgleich mit dem Verhalten der Software heranzuziehen

ist besonders in Randfällen hilfreich. Angenommen ein Online-Shop bietet einen kostenlosen Versand ab einem Mindestbestellwert an. Sobald der Preis den Wert von 25$ übersteigt ist der Versand gratis. Der Versand kostet im Normalfall 5$. Nun muss definiert werden, ob der Versand in dem Mindestbestelltwert miteinbezogen wird oder nicht. Wird er miteinbezogen, dann sorgt das dafür, dass ein Produkt, welches 20$ kostet einen kostenlosen Versand hat, weil der Mindestbestellwert zuzüglich dem Versandpreis 25$ beträgt. Ob dieses Verhalten gewünscht ist oder nicht, muss in der Spezifikation festgehalten werden. Wenn nun ein Bugticket dieser Art bei einem Softwareentwickler landet, dann muss er, wie auch bei der Implementierung, alle relevanten Dokumente für die weitere Entwicklung heranziehen.//

Neben dem Auffinden der relevanten Dokumente aus der Wissensdatenbank kann die Suche einen weiteren Mehrwert bieten, wenn sie auch gleich die relevante Stelle im Code findet, welche angepasst werden muss, um das Feature zu implementieren oder zu erweitern, oder um einen Bug zu beheben. *Feature Location* ist das Vorgehen, den Einstiegspunkt im Code zu finden, welcher für ein Feature verantwortlich ist. Wenn eine Software das Feature besitzt, Mails zu verschicken, dann könnte der Einstiegspunkt für dieses Feature eine Schnittstelle sein, welche den Inhalt und Empfänger der Email entgegennimmt. Der Inhalt wird anschließend in ein Mail-Template eingefügt, und die Mail wird verschickt. Wenn die Suchfunktion nun Feature Location ermöglicht, dann wird damit die Erweiterung des Features vereinfacht, weil der Software-Entwickler die anzupassende Stelle nicht manuell suchen muss.

Analog hierzu ist *Bug Localization* ein Verfahren, um die Quelle von Fehlern im Code einzugrenzen.

2.3. Bug Localization

Neben der Implementierung neuer Features ist es Aufgabe eines Softwareentwicklers bestehende Fehler in der Software zu korrigieren. Um einen Fehler zu korrigieren, ist es sinnvoll die Spezifikation heranzuziehen, um herauszufinden, wie sich die Software ordnungsgemäß verhalten sollte.

Wenn der Softwareentwickler nun ein Fehlerticket erhält, dann muss er die entsprechende Spezifikation zu diesem Fehlerticket finden können. Und idealerweise wird ihm durch die Suche sogar gleich die betroffene Stelle im Code angezeigt. Das ist Bug Localization.

2.4. Informationen über das Projektmanagement finden

Softwareentwickler sollten darüber Bescheid wissen, was in dem Projekt, in dem sie arbeiten, vor sich geht. Sie sollten über organisatorische Dinge Bescheid wissen, wie die Teamaufteilung und die Zuständigkeiten in den einzelnen Teams und

in dem gesamten Projekt. Das ist wichtig für eine gute Kommunikation und entsprechenden Wissensaustausch zwischen Kollegen. Außerdem sollten Softwareentwickler darüber Bescheid wissen, wann das nächste Release der Software ansteht. Diesen Termin müssen sie bei der Planung ihrer Arbeit berücksichtigen, damit sie auch rechtzeitig alle relevanten Features implementiert haben und alle kritischen Fehler behoben haben. Neben der Einhaltung von Terminen ist die Planung wichtig für die Motivation der Softwareentwickler. Denn Motivation entsteht durch die Wahrnehmung, sich einem Ziel zu nähern (Quelle: Flow - mihaly csikszentmihaly). Aus diesem Grund müssen die Softwareentwickler auch die Vision der Software verstehen und auch die übergreifende Vision des Unternehmens. Sie müssen verstehen, warum die Arbeit, welche sie erledigen so wichtig ist. Und sie müssen verstehen, für wen sie diese Arbeit tun. Auch diese Faktoren sind wichtig für eine hohe Motivation bei der Arbeit. Daher ist es so wichtig diese Informationen einfach zugänglich zu machen, also einfach auffindbar zu machen.

In Kapitel 6 wird aufgezeigt, wie die Anforderungen der Anwendungsfälle quantifizierbar gemacht werden können. Das wird dabei helfen nachzuvollziehen, inwieweit die Anforderungen der Anwendungsfälle erreicht wurden, welche genannt wurden.

3. Die Technologie von Confluence

Die Confluence Suche verwendet laut Dokumentation Apache Lucene.[1] Apache Lucene ist eine Bibliothek, welche Indizierungs- und Suchfeatures umfasst. Dabei verwendet Apache einen Volltext-Index und bietet Vector Space Models und BM25 an, um nach passenden Dokumenten zu suchen. Laut der Dokumentation von Apache Lucene, verwendet dieses einen Scoring Algorithmus, welcher sowohl auf VSM als auch auf Boolean Models basiert.[2] Im Information Retrieval sind Boolean Models jene Scoring Algorithmen, welche auf boolscher Algebra basieren. Boolean Models werden benötigt, um eine Boolean Search zu realisieren. Auf die Boolean Search wird im späteren Verlauf des Kapitels eingegangen. Aus der Dokumentation von Apache Lucene und aus dem genannten Paper geht ebensfalls hervor, dass die Vektoren des VSM mit tf-idf berechnet werden.

Dieses Kapitel hat den Zweck die Funktionsweise der Suchfunktion von Confluence zu erläutern. Dies ist notwendig, um später die Ergebnisse des Vergleichs der Suchfunktion von Confluence mit der neuen Suchfunktion zu verstehen. Um die Suchfunktion von Confluence zu verstehen, ist es zunächst notwendig einen Überblick über die wichtigsten Komponenten einer Suchfunktion zu bekommen. Daher stellt dieses Kapitel zuerst die einzelnen Komponenten einer Suchfunktion dar. Als nächstes werden die spezifischen Technologien erläutert, welche Apache Lucene verwendet. Das bedeutet, dass zuerst erklärt wird, was ein Volltext-Index ist. Anschließend werden die Scoring Algorithmen TF-IDF und BM25 erläutert. Zuletzt wird zusammenfassend erklärt, welche Suchmethoden mit den Technologien von Apache Lucene ermöglicht und in Confluence verwendet werden.

3.1. Die Architektur einer Suchfunktion

Dieses Kapitel gibt einen Überblick über die wichtigstes Komponenten einer Suchfunktion. Eine einfache Implementierung einer Suchfunktion kann aus drei Komponenten bestehen. Aus dem Crawling, dem Index und dem Scoring-Algorithmus. Das Crawling ist zuständig für das Finden von Dokumenten (Castillo 2005). Der Index speichert Informationen der Dokumente, und die Suche ist zuständig für das Verstehen der Nutzeranfrage und die Abfrage der relevantesten Informationen aus dem Index, sowie dessen Verarbeitung und Darstellung. Der Begriff Dokument wird hier verwendet, um die Dateien zu beschreiben, welche durch einen Crawler gesucht und durch den Index verarbeitet werden. Unter Dokumenten können auch eine Website verstanden werden, welche durch einen Webcrawler durchsucht

[1] `https://confluence.atlassian.com/doc/ranking-of-search-results-1188406620.html`

[2] `https://lucene.apache.org/core/2_9_4/scoring.html`

werden. Für die Implementierung der Suche wird ein Scoring-Algorithmus benötigt. Der Scoring-Algorithmus bestimmt, welche Dokumente am besten auf eine Sucheingabe passen.

Für die Implementierung einer Suchfunktion wird zunächst ein Datensatz von Dokumenten benötigt, welche über die Suchfunktion gefunden werden können. Dieser wird mithilfe eines Crawlers aufgebaut. Ein Crawler ist ein Algorithmus, welcher Techniken aus dem Natural Language Processing nutzt, um Informationen aus einem Dokument zu extrahieren (Khder 2021). Implementierungen können reguläre Ausdrücke verwenden, um die Informationen zu extrahieren, oder auch fortgeschrittenere Verfahren, wie Abstract Syntrax Trees. Im Falle von Websites kann der Crawler Hyperlinks zu weiteren Websites extrahieren. Damit kann der Algorithmus sukzessive den Datensatz von Dokumenten befüllen. Die neuen Dokumente werden durch den Index verarbeitet und wiederum auf neue Links analysiert. Dieses Verfahren kann beliebig lange und beliebig rekursiv durchlaufen werden, um den Index zu erweitern. Idealerweise, ohne Seiten dabei mehrfach zu durchlaufen. Neben dem Crawling können Indizes befüllt werden, indem eine Liste von Dokumenten übergeben werden, welche dem Index hinzugefügt werden sollen.

Die Indizierung von Dokumenten kann sowohl mit einer Volltext-Indizierung oder auch einer Vektor-Indizierung umgesetzt werden. Ein Volltextindex bestimmt den Score anhand von Ausschnitten aus dem Volltext. Wenn diese bestimmte Kriterien erfüllen, dann wird das Dokument bei der Suche gefunden, andernfalls nicht. Ein Vektorindex berechnet Vektoren anhand des Ursprungstextes. Ein Vektorindex wird auch als Vector Space Model (VSM) bezeichnet. Die Berechnung der Vektoren kann mithilfe von tf-idf, BM25, word2vec, Latent Semantic Embeddings oder auch mit Transformers erfolgen.

3.2. Volltext-Indizierung

Nachdem ein Überblick über die Komponenten einer Suchfunktion gegeben wurde, werden nun die Technologien erläutert, welche Apache Lucene bereitstellt. Apache Lucene verwendet einen Volltext-Index. Eine Möglichkeit zur Umsetzung einer Volltext-Indizierung ist der invertierte Index. Zur Generierung des invertierten Indexes müssen die zu indizierten Dokumente zunächst in einer Datenbank abgelegt werden. Ein invertierter Index wird generiert, indem alle Wörter extrahiert werden, welche in den Dokumenten vorkommen. Nun werden diese Wörter in eine Liste geschrieben, und jedem Wort wird zugeordnet, in welchem Dokument sich dieses Wort wiederfinden lässt. Diese Liste wird invertierter Index genannt, weil nicht die Wörter den Dokumenten zugeordnet sind, sondern die Dokumente den Wörtern. Es wird ebenfalls gespeichert, an welchen Stelle des Dokuments das Wort vorkommt, und auch in wie vielen Dokumenten ein Wort vorkommt.

Bei der Indizierung der Wörter besteht nun die Problematik, dass gleiche Wörter

in unterschiedlichen Formen existieren können. So stammen *Heizung* und *heizen* beide von dem gleichen Wortstamm *heiz* ab. Um bei der Indizierung Speicherplatz zu sparen, können Wörter auf diesen Wortstamm reduziert werden, damit sie als ein einziges Wort betrachtet werden können. Die Bildung des Wortstamms wird als Stemming bezeichnet. Beim Stemming kann es jedoch zu Overstemming und Understemming kommen. Overstemming bedeutet, dass zwei Wörter, die eigentlich nichts miteinander zu tun haben, also nicht semantisch gleich sind, den gleichen Wortstamm besitzen und als ein Wort betrachtet werden. Ein Beispiel hierfür sind die Wörter *Wand* und *wandere*, wie in *ich wandere*. Beide besitzen den Wortstamm *wand* und werden entsprechend als ein Wort betrachtet. Understemming bedeutet, dass zwei Wörter, die eigentlich etwas miteinander zu tun haben, also semantisch gleich sind, nicht den gleichen Wortstamm besitzen und dadurch als zwei verschiedene Wörter betrachtet werden. Ein Beispiel hierfür sind die Wörter *absorbieren* und *Absorption*, welche die Wortstämme *absorb* und *absorp* besitzen. Es gibt Techniken zur Vermeidung solcher Probleme, wie der Einsatz vollständiger morphologischer Analysekomponenten. Hierauf soll aber nicht weiter eingegangen werden.

3.3. TF-IDF

Da nun die Dokumente indiziert sind, kann ein Scoring-Algorithmus bestimmen, welche Dokumente die passendsten für eine gegebene Sucheingabe sind. Apache Lucene verwendet für das Scoring von Dokumenten die TF-IDF Metrik. TF-IDF untersucht, wie häufig Terme in einem Dokument vorkommen und wie charakteristisch diese Terme für dieses Dokument sind (Manning et al. 2019). Kommt ein gesuchter Term häufig in einem Dokument vor, dann erhöht dies den Score für dieses Dokument. Kommt dieser Term selten in anderen Dokumente vor, dann erhöht dies den Score ebenfalls. Umgekehrt ist der Score für ein Dokument niedriger, je seltener der gesuchte Term in dem Dokument vorkommt oder je häufiger der Term in allen anderen Dokumenten vorkommt. TF-IDF ist folgendermaßen definiert:

$$\text{tf-idf}_{t,d} = \text{tf}_{t,d} \times \text{idf}_t$$

$$tf(t, D) = \frac{\#(t,D)}{\max_{t' \in D} \#(t',D)}$$

$$\text{idf}_t = \log \frac{N}{\text{df}_t}$$

Der tf-Teil steht für *term frequency* und wird berechnet, indem für das jeweilige Dokument bestimmt wird, wie häufig das Wort in dem Dokument vorkommt. Damit die Metrik nicht abhängig von der Länge des Dokumentes ist, wird dieser Wert durch die insgesamte Anzahl der Vorkommnisse des Wortes dividiert. Damit ist diese Metrik relativ zur insgesamten Anzahl der Vorkommnisse des Wortes, und

nicht absolut. Der idf-Teil steht für *inverse document frequency*. Er wird berechnet indem die insgesamte Anzahl der Dokumente durch die Anzahl der Dokumente dividiert wird, welche das Wort enthalten. Das Ergebnis des dividierens wird an die Logarithmus-Funktion übergeben, sodass am Ende der idf-Wert berechnet wurde. Die beiden Werte werden miteinander multipliziert. Das Ergebnis ist die tf-idf-Metrik.

Aufbauend auf der TF-IDF Metrik wurde die BM25 Metrik entwickelt, welche Apache Lucene ebenfalls bereitstellt.

3.4. Die Suchmethoden von Confluence

Dieses Kapitel stellt die Arten von Suchanfragen vor, welche durch Apache Lucene ermöglicht werden. Es ist wichtig diese zu kennen, um das Qualitätskriterium später bei der Implementierung erfüllen zu können. Es gibt die gängigen Suchalgorithmen Keyword Search, Phrase Search, Boolean Search, Field Search.

Eine Keyword Search durchsucht Dokumente nach der Sucheingabe des Nutzers. Die Eingabe wird dabei nicht als Ganzes betrachtet, sondern jedes Wort einzeln. Für jedes Keyword werden Dokumente als Ergebnis angezeigt, wenn dieses in dem Dokument vorhanden ist. Wenn ein Dokument mehrere der Keywords beinhaltet, wird dessen Relevanz höher eingeschätzt als für Dokumente, welche weniger Keywords enthalten. Dokumente mit höherer Relevanz werden weiter oben in der Ergebnisliste angezeigt.

Eine Phrase Search ist die Suche nach Textausschnitten in Dokumenten. Hier werden nicht mehrere Keywords einzeln betrachtet, sondern die gesamte Eingabe in das Suchfeld als eine Einheit. Es reicht also nicht mehr aus, dass ein Dokument eines der Wörter enthält. Es muss die gesamte Sucheingabe als ein String enthalten sein.

Die Boolean Search bietet die Möglichkeit einen boolschen Ausdruck als Sucheingabe zu machen. Ein Beispiel dafür ist die Sucheingabe *Dokumentation AND Angular*. Die Sucheingabe bedeutet, dass die Suchfunktion nur Dokumente als Ergebnis darstellen soll, welche beide Keywords Dokumentation und Angular enthalten. Die Boolean Search kann auch eine Phrase, wie bei der Phrase Search, beinhalten: *"Dokumentation von Software" AND Angular*. In diesem Beispiel werden nur Dokumente als Ergebnis nur angezeigt, wenn sie den gesamten String *Dokumentation von Software* enthalten, sowie das Keyword *Angular*. Bei einer Boolean Search können die boolschen Operatoren *AND*, *OR*, *NOT* beliebig kombiniert werden.

Eine Field Search sucht Dokumente anhand von Attributen. Der Nutzer kann diese Attribute auswählen. Wenn der Nutzer beispielsweise ein Dokument sucht,

welches am 01.01.2005 erstellt wurde, dann kann die Eingabe der Suche so aussehen: *erstelldatum: 01.01.2005*. Es können beliebig viele Attribute verwendet werden, um die Suche einzugrenzen. Neben der Verwendung der Attribute für die Suche selbst, können die Attribute komplementär zu einer anderen Art von Suche verwendet werden. So kann eine Suchfunktion Buttons bereitstellen, über welche Filter festgelegt werden. Nun kann eine Keyword Search durchgeführt werden, aber die gefundenen Dokumente werden mithilfe der Filter weiter eingeschränkt. Neben diesen gängigen Suchalgorithmen gibt es weitere Suchalgorithmen, wie die strukturierte Suche und die semantische Suche.

4. Konzeption der Suchfunktion

4.0.1. Vektor-Indizierung

Neben einer Volltext-Indizierung können Dokumente in Form von Vektoren indiziert werden. Dazu werden Dokumente zunächst, wie auch bei der Volltext-Indizierung gecrawlt. Anschließend durchlaufen die Inhalte der Dokumente ein Preprocessing. Dieses kann je nach Implementierung variieren. Das Kapitel *Einspielen der Daten in Weaviate* beschreibt, wie in der hier aufgeführten Implementierung das Preprocessing durchgeführt wird. Das Preprocessing hat den Zweck die Daten an das Schema der Datenbank anzupassen und die Qualität der Daten zu erhöhen. Außerdem sorgt es für eine kürzere Indizierungszeit.

Nach dem Preprocessing werden durch einen Transformer für die Inhalte der Dokumente Vektoren berechnet. Ein Transformer wird mithilfe von Trainingsdaten darauf trainiert, Vektoren für Wörter zu generieren. Das trainierte Modell wird nach dem Preprocessing durchlaufen. Zuletzt werden die Daten in einer Vektordatenbank gespeichert. Eine Vektordatenbank speichert Daten, wie eine dokumentenbasierte Datenbank. Dort werden nun sowohl die rohen Daten als auch die Vektoren gespeichert, welche von dem Transformer berechnet wurden. Die Vektoren haben den Vorteil, dass die Daten in der Datenbank nicht linear gespeichert sind. Sie sind in einem n-dimensionalen Raum gespeichert, mit dessen Hilfe die semantische Nähe zwischen Dokumenten ausgedrückt werden kann. Das funktioniert auf die gleiche Weise, wie bereits in dem Kapitel *Semantic Search* beschrieben.

4.1. Sentence Embeddings

(Reimers und Gurevych 2019)

4.1.1. Word Embeddings

Word Embeddings sind eine Vektorrepräsentation von Wörtern. Zweck von Word Embeddings ist es, semantische Ähnlichkeiten zwischen Wörtern zu verstehen. Eine einfache Implementierung von Word Embeddings könnte sein, jedem Wort eine zufällige Zahl zuzuordnen. Wenn die Zahlen zufällig sind, werden damit allerdings keine semantischen Ähnlichkeiten kodiert. Um herauszufinden, welche Wörter eine semantische Ähnlichkeit besitzen.

4.1.2. word2vec

Word2vec ist ein zwei-Layer neuronales Netzwerk, welches die Semantik von Wörtern lernt. Für jedes Wort, für welches eine Vektorrepräsentation bestimmt werden soll, benötigt das neuronale Netzwerk einen Input. Das erste Layer des neuronalen Netzwerkes besteht aus Aktivierungsfunktionen. Jede Node des Layers summiert die Inputs. Die Weights der Aktivierungsfunktionen sind am Ende die Vektorrepräsentationen der Wörter. Die Anzahl der Nodes im Aktivierungslayer bestimmt die Anzahl der Dimensionen im Vektor. Das nächste Layer summiert wiederum die Werte aus dem Aktivierungslayer und wendet eine SoftMax Funktion an. Der Output des neuronalen Netzwerkes besteht, so wie der Input, aus so vielen Nodes, wie es Wörter gibt, welche mit dem Netzwerk repräsentiert werden sollen. Der Output gibt an, mit welcher Wahrscheinlichkeit jedes der Wörter das Folgewort des gegebenen Inputs ist.

Das neuronale Netzwerk wird mithilfe von Backpropagation trainiert. Dazu wird Training Data benötigt. Dieses enthält Sätze, auf welche das neuronale Netzwerk trainiert werden soll. Aufgabe des neuronalen Netzwerkes ist es, auf Basis eines gegebenen Wortes das nächste Wort im Satz zu bestimmen. Für ein einfaches Beispiel könnte das Trainingsdatenset aus zwei Einträgen *NLP ist interessant* und *NLP ist komplizit* bestehen. Da das Trainingsdatenset insgesamt aus vier verschiedenen Wörtern besteht, hat das neuronale Netzwerk genau vier Inputs. Wenn nun das Folgewort für das Wort *NLP* trainiert werden soll, dann bekommt das neuronale Netzwerk als Input die Werte *1, 0, 0, 0*. Die Eins bedeutet, dass das Wort NLP aktiviert ist. Die Nullen bedeuten, dass die anderen Wörter nicht aktiviert sind. Nun muss das neuronale Netzwerk für diesen Input den Output-Wert berechnen. Für die Backpropagation wird die Cross-Entropy Loss Function verwendet. Das Ergebnis ist, dass das trainierte neuronale Netzwerk für das Wört *NLP* lernt, dass das nächste Wort mit der Wahrscheinlichkeit eins das Wort *ist* ist. Die gelernten Weights bilden einen Vektor, welcher in einer Vektordatenbank gespeichert werden kann. Das Ergebnis ist eine Vektordatenbank, welche Informationen darüber enthält, welche Wörter semantisch ähnlich sind, und welche nicht.

4.1.3. seq2seq

Ein seq2seq-Modell soll eine Sequenz auf eine andere Sequenz anhand von erlernten Regeln mappen. Dazu wird die Input-Sequenz zunächst in eine modell-interne Sequenz encodiert, der Context Vector, und anschließend in die Output-Sequenz dekodiert. Wenn eine Wortsequenz in eine andere Wortsequenz umgewandelt werden soll, dann müssen die Input-Wörter zunächst durch ein Word Embedding Layer verarbeitet werden, welches die Wörter in Word Embeddings konvertiert. Auf das Word Embedding Layer folgen eine beliebige Anzahl von LSTM-Layers. Die Anzahl der Nodes der LSTM-Layer entspricht der Anzahl der Nodes im Word Embedding Layer. Zusätzlich kann jede Node in jedem LSTM-Layer mehrere LSTM Netzwerke besitzen. Die Weights und Biases der LSTM Netzwerke bilden

den Output und damit den Context Vector. Die bisher beschriebene Architektur wird als Encoder bezeichnet. Die LSTM Netzwerke des Encoders werden im Decoder gespiegelt. Die Outputs des oberen LSTM Layers des Decoders werden in ein Fully Connected Layer gegeben und durchlaufen eine Softmax Funktion. Der Output des Decoders sind die Wahrscheinlichkeiten, mit welchen die möglichen Wörter der Ziel-Sequenz dem Input entsprechen.

Der Context Vector kann in einer Vektordatenbank gespeichert werden, so wie Word Embeddings in einer Vektordatenbank gespeichert werden können. Das hat zur Folge, dass die Ähnlichkeit von Sätzen genauso wie die Ähnlichkeit von Wörtern ermittelt werden kann.

4.1.4. Scoring Algorithmen

4.2. Cosine Similarity

4.3. Hierarchical Navigable Small Worlds (HNSW)

HNSW ist ein Nearest Neighbor Suchalgorithmus. Der Algorithmus findet die ähnlichsten Nachbar-Vektoren zu einem gegebenen Input-Vektor. Um den Algorithmus umzusetzen werden Skip Lists und Navigable Small Worlds verwendet. Skip Lists sind klassische Linked Lists, welche aus mehreren Schichten besteht. Die unterste Schicht entspricht der originalen Linked List und beinhaltet alle Daten. Jedes höhere Layer beinhaltet nur ein Subset der Nodes des darunterliegenden Layers. Wenn nun eine Suche in der Skip List durchgeführt wird, dann wird das oberste Layer zuerst durchsucht. Das oberste Layer wird sequenziell durchlaufen. Es wird bei dem ersten Element des obersten Layers begonnen. Solange das aktuelle Element kleiner ist als das gesuchte Element, wird das nächste Element untersucht. Wenn das Element n dem gesuchten Element entspricht, dann wurde das gesuchte Element gefunden und wird zurückgegeben. Wenn das nächste Element n+1 größer ist als das gesuchte Element, dann wird das aktuelle Element n in dem aktuellen Layer gefunden. Da dieses Element noch nicht dem gesuchten Element entspricht, wird nun im tieferen Layer weitergesucht, in welchem mehr Elemente vorhanden sind. Denn das gesuchte Element liegt zwischen dem gefundenen Element n und dem untersuchten Element n+1. Der gleiche Algorithmus wird rekursiv für jedes Layer durchgeführt, bis das unterste Layer erreicht wurde. Wurde am Ende des untersten Layers kein Element gefunden, dann konnte das Element in der Skip List nicht gefunden werden (Pugh 1990).

4.4. Suchalgorithmen

Im Folgenden werden zuerst gängige Suchalgorithmen erklärt, welche in nahezu jeder Suchfunktion zu finden sind. Danach wird im speziellen auf die strukturierte Suche und die semantische Suche eingegangen. Die semantische Suche wird später bei der Implementierung einer Suchfunktion verwendet werden.

4.4.1. Die semantische Suche

Unter einer semantischen Suche wird im Allgemeinen verstanden, dass die Suche nicht nur eine syntaktische Suche einer Zeichenkette ist, sondern auch Techniken, wie technische Analysen verwendet, um die Bedeutung der Sucheingabe nachzuvollziehen (Dengel 2012). Eine semantische Suche hat den Zweck, die Ähnlichkeit und Beziehungen zwischen Wörtern zu verstehen. Sie kennt Homonyme, Synoyme und Antonyme von Wörtern. So wird durch sie beispielsweise die Ähnlichkeit von den Wörtern *rollout* und *deployment* abgebildet, und dass diese Wörter oft im gleichen Kontext verwendet werden.

Die technische Umsetzung einer semantischen Suche kann auf verschiedene Arten erfolgen. Zum einen besteht die Möglichkeit ein explizites semantisches Netz heranzuziehen, und so die Zusammenhänge von Wörtern abzubilden. Ein semantisches Netz ist eine Menge aus Aussagen in der Form *Subjekt Prädikat Objekt*. Anhand dieser Aussagen wird ein Graph aus Beziehungen zwischen Wörtern hergestellt (Lehmann o. D.).

Das hat den Vorteil, dass ein solches explizites semantisches Netz von Menschen erstellt wurde, und damit eine hohe Qualität der Daten einhergeht. Denn an der Erstellung von semantischen Netzen sind mehrere Menschen beteiligt, welche zuerst einen Konsens über die Eigenschaften und Zusammenhänge von Wörtern schaffen müssen. Der Nachteil dieser Methode ist der große Arbeitsaufwand, welcher mit der Erstellung eines solchen semantischen Netzes einhergeht. Ein weiterer Nachteil ist die mögliche Unvollständigkeit, welche ein solches semantisches Netz besitzen könnte. Eine Wissensdatenbank, welche für ein Projekt erstellt wurde, kann Definitionen von Begriffen beinhalten, welche in allgemeinen semantischen Netzen nicht vorhanden sind. Bei der Umsetzung einer semantischen Suche mithilfe eines semantischen Netzes müsste also zuerst ein allgemeines semantisches Netz herangezogen werden, beispielsweise von DBPedia. Anschließend müsste dieses semantische Netz um die neuen Definitionen, welche nur im Kontext des Projektes gelten, erweitert werden.

Eine weitere Möglichkeit zur Umsetzung einer semantischen Suche ist die Verwendung von Transformers und Vektordatenbanken. Um zu verstehen, welche Wörter kontextuell zusammengehören, werden hier die Wörter in einem n-dimensionalen Raum positioniert. Wörter, die sich sehr ähnlich sind, also im gleichen Kontext verwendet werden, haben in diesem n-dimensionalen Raum eine geringe Distanz zueinander. Wörter, die sich eher unähnlich sind, wie *"rollout" und "API"*, haben eine größere Distanz. Der Vorteil einer semantischen Suche ist, dass der genaue Begriff, welcher gesucht wird nicht bekannt ist. Wenn sich der Nutzer also über ein Thema informieren möchte, mit welchem er nicht gut vertraut ist, dann kann die semantische Suche hilfreich sein. Denn der Nutzer kann nun einen Begriff eingeben, der zu dem Thema passt, und den er kennt. Er findet anschließend Dokumente, welche vielleicht nicht genau diesen Begriff beinhalten, aber welche

thematisch dennoch ähnlich sind. Genau dieser Vorteil soll bei der Implementierung später genutzt werden.

Um eine semantische Suche zu implementieren, werden die Technologien von Transformern und Vektordatenbanken verwendet. Ein Transformer erhält als Input eine große Menge an Text und mappt die einzelnen Wörter auf einen Vektor einer beliebigen Länge. Der Vektor, der am Ende herauskommt, beschreibt die Position des Wortes in dem n-dimensionalen Raum. Der Vektor beschreibt gewissermaßen, wie stark ein Wort in eine abstrakte Kategorie einzuordnen ist. Jeder Wert im Vektor entspricht einer Kategorie. Mithilfe der Vektoren können verschiedene Wörter hinsichtlich ihrer Ähnlichkeit analysiert werden. Ähnliche Wörter habe eine große räumliche Nähe, während zwei Wörter, die in vollkommen unterschiedlichen Kontexten verwende werden eine sehr große Distanz im Raum besitzen. Nehmen wir für ein Beispiel einen dreidimensionalen Raum an. Die X-Achse ist beschriftet mit dem Wort „Tier“, die Y-Achse ist beschriftet mit dem Wort „Computer“ und die Z-Achse ist beschriftet mit dem Wort „Mensch“. Nun geben wir einem Transformer das Wort „Katze“, und der Transformer berechnet einen dreidimensionalen Vektor, welcher das Wort „Katze“ im Raum positioniert. Weil eine Katze ein Tier ist, ist der X-Wert des Vektors eins. Der Wert eins bedeutet, dass das Wort vollständig zu dieser Kategorie gehört. Da eine Katze überhaupt nichts mit einem Computer zu tun hat, ist der Y-Wert des Vektors 0.

Nun ist eine Katze kein Mensch, aber eine Katze ist ein Haustier von Menschen. Es ist denkbar, dass die Wörter Katze und Mensch oft im gleichen Kontext verwendet werden, sodass der Wert bei 0,3 liegen könnte. Damit der Transformer einen Vektor berechnen kann, braucht er eine Menge Daten. Diese Daten erhält er aus vielen Texten. Werden zwei Wörter oft im gleichen Text genannt oder kommen zwei Wörter in vielen Texten sehr nahe beieinander vor, dann geht der Transformer davon aus, dass die beiden Wörter ähnlich sind, und berechnet ähnliche Vektoren. Zuvor müssen die Texte allerdings bereitgestellt werden. Dazu kann beispielsweise das Internet gecrawlt werden. Die Ergebnisse des Transformers werden in einer Vektordatenbank gespeichert. Eine Vektordatenbank ist eine Datenbank, welche Vector Embeddings, also ein Objekt als Key und dessen Vektor als Value speichert. Bei dem Objekt kann es sich um Wörter handeln, dann wird auch von Word Embeddings gesprochen. Es können aber auch Daten andere Daten, wie Bilder, Videos oder Audio gespeichert werden. Der Zweck von Vektordatenbanken ist es, Daten nicht einfach linear zu speichern, sondern in einem Raum. Die Distanz zwischen zwei Einträgen in diesem Raum beschreibt dessen Ähnlichkeit. Genau diese Informationen nutzen semantische Suchen.

4.5. Wahl der Suchfunktion

Die vergangenen Kapitel haben verschiedene Indizierungs- und Suchalgorithmen dargestellt. Jeder dieser Algorithmen hat Vor- und Nachteile. Diese werden in die-

sem Kapitel betrachtet, um die Entscheidung für die verwendeten Technologien der neuen Suchfunktion nachvollziehbar zu machen.

Die Confluence Suche verwendet laut Dokumentation Apache Lucene.[1] Laut der Dokumentation von Apache Lucene, verwendet dieses einen Scoring Algorithmus, welcher sowohl auf VSM als auch auf Boolean Models basiert.[2] Im Information Retrieval sind Boolean Models jene Scoring Algorithmen, welche auf boolscher Algebra basieren. Aus der Dokumentation von Apache Lucene und aus dem genannten Paper geht ebensfalls hervor, dass die Vektoren des VSM mit tf-idf berechnet werden.

In einem Paper von Choudhary et. al. wird ein Document Retrieval System entwickelt (Choudhary et al. 2020). Das Document Retrieval System verwendet Bert zur Generierung von Embeddings. Der Scoring Algorithmus des Systems kombiniert Bert und tf-idf, um den Score zu berechnen. Laut dem Paper bietet eine Kombination aus tf-idf und Bert zur Implementierung eines Document Retrieval Systems signifikante Performance Verbesserungen gegenüber einem Document Retrieval System, welches lediglich tf-idf verwendet. Die Performance Verbesserung wird in dem Paper an dem MS Marco Datensatz gemessen.[3]

In einem Paper von Karpukhin et. al. wird Open-Domain Question Answering untersucht (Karpukhin et al. 2020). Dazu wird ein Dense Passage Retriever entwickelt. Ein Dense Passage Retriever bestimmt den Textabschnitt eines gegebenen Textes, welcher mit größter Wahrscheinlichkeit eine Antwort auf eine gestellte Frage beinhaltet. Der Input für einen Dense Passage Retriever sind Dokumente, welche zuvor mithilfe eines Scoring Algorithmus aus einem Index extrahiert wurden. Das Paper zeigt, dass das entwickelte Passage Retrieval System besser darin ist relevante Textabschnitte zu finden als der BM25 Algorithmus. Das Paper nennt die semantische Verknüpfung von Synonymen und Paraphrasierungen mit unterschiedlichen Tokens als Vorteil gegenüber BM25 und auch TF-IDF.

Die semantische Suche ist bei dem Einspielen der Daten in die Datenbank langsamer, weil zunächst die Vektoren für die Daten generiert werden müssen. Die Zeit für die Abfrage von Daten aus einem Vector Space Model wächst mit wachsender Anzahl von Datenpunkten nicht so schnell an, wie bei einem Volltext-Index. Bei einem Volltext-Index müssen die übergebenen Keywords mit jedem Datensatz abgeglichen werden. Bei einem Vector Space Model sind zwar mit steigender Zahl von Datensätzen mehr Punkte im Raum vorhanden, aber es müssen nicht für jeden Punkt Strings verglichen werden. Der Algorithmus, welcher für die Ermittlung der passendsten Datenpunkte im Vector Space Model verwendet wird, lautet Approximate Nearest Neighbor (ANN). Dieser basiert auf linearer Algebra.

Bei der Implementierung sollen dabei keine semantischen Netze verwendet werden, sondern ein Vector Space Model.

[1] `https://confluence.atlassian.com/doc/ranking-of-search-results-1188406620.html`
[2] `https://lucene.apache.org/core/2_9_4/scoring.html`
[3] `https://microsoft.github.io/msmarco/`

Neben den direkten Vorteilen einer semantischen Suche gegenüber Suchfunktionen, wie eine Keyword Search, gibt es weitere indirekte Vorteile eine semantischen Suche auf Grundlage von Vector Space Models. Vector Space Models sind Multimodal. Es können also in einem Vector Space Model nicht nur Informationen über ein Medium, wie Fließtext, gespeichert werden, sondern neben Fließtext können gleichzeitig auch andere Medien, wie Code, Bilder, Audiodateien etc. gespeichert werden. Und anders als bei einer gewöhnlichen Datenbank, in welcher mehrere verschiedene Medien gespeichert werden können, beispielsweise eine relationale Datenbank, können Vector Space Models diese verschiedenen Medien miteinander semantisch verknüpfen.

Wenn ein Softwareentwickler ein bestehendes Feature anpassen muss, dann muss er zunächst den Einstiegspunkt im Code kennen. Sind keine weiteren Informationen vorhanden, dann muss der Softwareentwickler dazu den Code manuell durchsuchen. Mithilfe von Techniken aus dem Natural Language Processsing, wie Latent Semantic Indexing (LSI), Latent Dirichlet Allocation (LDA) und Vector Space Models kann dieser Prozess automatisiert werden (Dit et al. 2011). Mithilfe eines gemeinsamen Index von den Inhalten der Wissensdatenbank und dem Quellcode lässt sich ein Zusammenhang zwischen Feature-Spezifikation und Klassen im Quellcode herstellen. Die Herangehensweise erfordert eine Möglichkeit beide Datenquellen auf die gleiche Art und Weise zu indizieren. Dazu werden LSI und Vector Space Models verwendet.„Information retrieval models for recovering traceability links between code and Documentation“

Indem die Codebase, welche zu der Wissensdatenbank in Verbindung steht, ebenfalls indiziert wird, kann eine Traceability zwischen Code und Wissensdatenbank hergestellt werden. Damit können Feature Location und Bug Localization umgesetzt werden. So kann die Suchfunktion nicht nur zum Durchsuchen der Wissensdatenbank verwendet werden, sondern auch zum Durchsuchen der Codebase. Und wenn in der Suchfunktion nach einem Feature gesucht wird, dann kann neben der Spezifikation in der Wissensdatenbank auch gleich der Einstiegspunkt im Code gefunden werden. Damit verbessert sich wiederum die Wartbarkeit der Software, da das Ändern von Features oder korrigieren von Bugs erleichtert wird.

4.6. Weitere Ansätze

4.6.1. Retrieval Augmented Generation

Neben einer Suchfunktion können System entwickelt werden, welche eine Question-Answering Funktionalität bieten. Eine Question-Answering Funktionalität nimmt eine Frage des Nutzers entgegen, und liefert eine Antwort. Solche Systeme basieren auf Retrieval Augmented Generation. Das bedeutet, dass auf Grundlage der Frage des Nutzers zuerst eine Suche durchgeführt wird. Diese Suche liefert relevante Dokumente zum Beantworten der Frage. Nun werden die, für die Frage relevantesten, Stellen der Dokumente mithilfe eines Large Language Models extrahiert

und umformuliert. Das Ergebnis ist eine Antwort auf die Frage des Nutzers.

4.6.2. Document Classification und Filter verwenden

TODO:

4.6.3. Verbesserung der Suchfunktion durch Traceability (insb. für Feature Location und Bug Localization)

TODO:

4.6.4. Inhaltsverzeichnisbezogen durchsuchen

TODO:

5. Implementierung der neuen Suchfunktion

In diesem Kapitel wird die Implementierung einer alternativen Suchfunktion zu der Suchfunktion von Confluence dargestellt. Diese Implementierung wird anschließend mithilfe des erläuterten Versuchsaufbaus mit der bestehenden Suche von Confluence verglichen. Die Implementierung verwendet eine Vektordatenbank, welche eine semantische Suche ermöglicht. Das Filtern nach bestimmten Properties ermöglicht die Vektordatenbank ebenfalls. Es wird die Vektordatenbank Weaviate verwendet. Der Vergleich zeigt, ob eine semantische Suchfunktion *besser* ist als die Confluence-Suche.

5.1. Aufsetzen der Vektordatenbank Weaviate

Für das Aufsetzen von Weaviate werden zwei Komponenten benötigt. Zum einen die Vektordatenbank selbst. Zum anderen ein Transformer, welcher Text entgegennimmt, und diese in Vektoren umwandelt, sodass diese in der Datenbank gespeichert werden können. Um die beiden Komponenten aufzusetzen wird Docker verwendet. Das entsprechende docker-compose-yml File ist im Anhang zu finden. Hier werden die beiden Komponenten definiert. Unter *t2v-transformers* wird der Transformer konfiguriert. Es wird das Image eines bereits vortrainierten Transformers verwendet. Unter *weaviate* wird die Datenbank konfiguriert. Hier wird mithilfe von den Environment-Variablen $DEFAULT\_VECTORIZER\_MODULE$, $ENABLE\_MODULES$ und $TRANSFORMERS\_INFERENCE\_API$ konfiguriert, welcher Transformer verwendet werden soll. So wird konfiguriert, dass der Transformer der anderen Komponente verwendet werden soll.

Mithilfe der Dependency *io.weaviate:client:4.0.1* wird die Client API von Weaviate verwendet. So wird nun eine Verbindung zu der Vektordatenbank aufgebaut. Diese beinhaltet zu diesem Zeitpunkt noch keine Daten. Bevor die Daten in die Datenbank eingespielt werden, muss das Schema der Daten angegeben werden. Der entsprechende Code ist ebenfalls im Anhang zu finden. Es wird eine Klasse *Document* definiert. Diese Klasse beinhaltet die Properties *documentUrl*, *h1*, *h2* und *p*. Es werden in dieser Klasse also die URL des Dokuments, sowie die Inhalte aller h1-, h2 und p-Tags gespeichert. Außerdem wird als Vectorizer *text2vec-transformers* konfiguriert.

5.2. Einspielen der Daten in Weaviate

Um nun die Daten aus Confluence in Weaviate einzuspielen, werden zuerst die Daten aus Confluence exportiert. Beim Export von Confluence Seiten werden HTML-Dateien generiert. Es werden keine CSS-, JavaScript- oder Bilddateien generiert. Hierdurch entstehen bei der Durchführung der Studie Probleme, auf welche später eingegangen wird. Die HTML-Dateien werden preprocessed, um dem zuvor definierten Schema zu entsprechen. Es werden zuerst mithilfe von regulären Ausdrücken alle Inhalte von h1-, h2- und p-Tags herausgefiltert. Anschließend werden Punctuations und Stopwords entfernt und die Inhalte druchlaufen einen Tokenizer und einen Stemmer. Punctuations sind Zeichen, wie die folgenden: *.,;:*. Stopwords sind Wörter, welche für einen Leser notwendig sind, aber für die Verarbeitung durch einen Algorithmus als unwichtig erachtet werden (Sarica und Luo 2021). Beispiele für Stopwords sind *aber, denn, der*. Ein Tokenizer trennt einen Text in einzelne Wörter auf. Aus einem Text, wie *das deployment erfolgt durch ein bash-skript* wird also ein Array, welches folgendermaßen aussieht:

$$[\text{"}das\text{"}, \text{"}deployment\text{"}, \text{"}erfolgt\text{"}, \text{"}durch\text{"}, \text{"}ein\text{"}, \text{"}bash\text{"}, \text{"}skript\text{"}].$$

Ein Stemmer bestimmt den Wortstamm für die einzelnen Wörter auf Basis von Grammatikregeln. Konkret verwendet die Software den Porter-Stemmer-Algorithmus. Aus dem Wort *deployment* wird dadurch beispielsweise *deploy*. Duplikate von Wörtern werden anschließend verworfen. Nachdem die Inhalte dieses Preprocessing durchlaufen haben, werden sie in die Datenbank eingespielt.

Im Kapitel *Vektorindizes* wurde bereits von Performancegründen gesprochen, aus denen das Preprocessing durchgeführt wird. Da nun die einzelnen Schritte des Preprocessings erklärt wurde, kann auch erklärt werden, warum das Preprocessing die Performance beim indizieren erhöht. Zuallererst werden viele Wörter gänzlich verworfen, weil sie Stopwords sind. Durch den Stemmer werden anschließend ähnliche Wörter zusammengruppiert. Die Wörter *Heizung* und *heizen* stammen beide vom gleichen Wortstamm *heiz*. Das bedeutet, dass aus zwei verschiedenen Wörtern, welche beide indiziert werden müssen, ein einziges Wort gemacht wird. Denn am Ende werden Duplikate, wie bereits erwähnt, verworfen. Die Anzahl der Wörter, welche indiziert werden müssen, wird dadurch reduziert, und damit auch die Last auf dem Transformer, welche die Vektoren für die Wörter berechnen muss.

5.3. Verwendung der Suchfunktion von Weaviate

Weaviate verwendet eine API, welche GraphQL queries entgegennimmt. Um eine Suche durchzuführen muss ein GET-Request durchgeführt werden. Nun muss angegeben werden, welche Klasse aus der Datenbank abgefragt werden soll. In diesem Fall *Document*.

5.3.1. Verwendung der Semantische Suche

Um die semantische Suche zu verwenden, muss die NearText Funktion verwendet werden.

6. Evaluationsmethoden und -Kriterien

Um zu verifizieren, dass die Implementierung ihre gewünschte Wirkung erzielt, müssen zuerst Methoden und Kriterien zur Messung herangezogen werden. Karl Popper gilt als Begründer des kritischen Rationalismus. Es war der Anfang von wissenschaftlich durchgeführten Experimenten. Bei einem Experiment wird zunächst eine Hypothese aufgestellt. Diese wird anschließend durch das Experiment untersucht. Ein solches Experiment soll später für die genannten Anwendungsfälle erstellt werden. Dazu sollen für die Anwendungsfälle Sucheingaben erstellt werden. Für jede Sucheingabe wird definiert, welche Dokumente als Ergebnis erwartet werden. Die Hypothese: Mit der implementierten Suchfunktion werden die erwarteten Dokumente *besser* gefunden als mit der bisherigen Suchfunktion. Diese Hypothese muss nun quantifizierbar und messbar gemacht werden. Die subjektive Wahrnehmung reicht nicht für eine wissenschaftliche Arbeit aus.

Die Formulierung der Hypothese ist für sich genommen zu unspezifisch. Es muss geklärt werden, wann eine Information *besser* zu finden ist. Im Folgenden werden Precision, Recall und F-Maß, sowie Qualitätskriterien gemäß ISO/IEC 9126 herangezogen, um Suchfunktionen anhand dieser Eigenschaften vergleichbar zu machen. Anhand dessen wird hergeleitet, was in diesem Kontext *besser* bedeutet. Im Anschluss werden diese Eigenschaften in einem Versuchsaufbau verwendet.

6.1. Precision, Recall und F-Maß

Zur Evaluation der Suchfunktionen werden später die statistischen Messwerte Precision und Recall verwendet. Die Messwerte beschreiben, inwieweit eine Hypothese zutrifft. Wenn also die Hypothese ist, dass ein bestimmtes Dokument gefunden wird, dann ist der Precision-Wert das Verhältnis zwischen allen gefundenen Dokumenten und den gefundenen Dokumenten, die tatsächlich relevant sind. Der Wert lässt sich im Kontext der Suche nach Dokumenten wie folgt definieren (Sirotkin 2012):

$$Precision = P = \frac{gefundenerelevanteDokumente}{gesamteAnzahlgefundenerDokumente}$$

Der Recall-Wert gibt wiederum an, wie viele von den tatsächlich relevanten Dokumenten auch gefunden wurden. Er lässt sich in diesem Kontext wie folgt definieren (Sirotkin 2012):

$$Recall = R = \frac{gefundenerelevanteDokumente}{gesamteAnzahlrelevanterDokumente}$$

Es ist schwierig beide Werte zu optimieren, da der Precision-Wert versucht die Anzahl der gefundenen Dokumente einzugrenzen und der Recall-Wert versucht die Anzahl der gefundenen Dokumente zu erweitern. Das F1-Maß fasst beide Werte zu einem neuen Wert zusammen (Sirotkin 2012):

$$F_1 = 2\frac{PR}{P+R}$$

Neben der Verwendung des F1-Maß ist es wichtig sich Gedanken darüber zu machen, welcher der beiden Messwerte wichtiger ist. In diesem Fall ist es sinnvoll eher die Precision zu optimieren. Denn, wenn ein Dokument gesucht wird, aber überhaupt nicht gefunden werden kann, dann erfüllt die Suchfunktion nicht ihren Zweck. Wenn die Suchfunktion irrelevante Dokumente darstellt, kann sie trotzdem ihren Zweck erfüllen, solange die relevantesten Dokumente zuerst in der Liste der Ergebnisse dargestellt wird. Dieser Faktor gilt auch bei der Implementierung zu berücksichtigen.

6.2. Versuchsaufbau

Für den Versuchsaufbau werden Dokumente benötigt, welche durchsucht werden können. Dazu sollen Daten aus einem echten Projekt verwendet werden. Da die Informationen zu dem Projekt nicht veröffentlicht werden dürfen, müssen die Sucheingaben und Ergebnisse so eingeschränkt werden, dass keine projektspezifischen Informationen preisgegeben werden. Die Daten sind bereits in einem Confluence Space vorhanden, dessen Suche als Benchmark für die neu implementierte Suchfunktion verwendet wird. Die Implementierung der neuen Suchfunktion und die Befüllung dessen Datenbank mit den gleichen Daten, wie die Confluence Suche, wird in dem Kapitel *Implementierung* näher erklärt.

Wie zuvor erwähnt, müssen nun Sucheingaben erstellt werden, welche in beiden Suchen eingegeben werden können. Zum Vergleich der beiden Suchfunktionen werden für jede Sucheingabe die obersten fünf Ergebnisse betrachtet. Für jedes Dokument in diesen Ergebnissen, welches laut Experimentaufbau als Ergebnis erwartet wird, erhält die Suchfunktionen einen Hit. Die Hits werden für beide Suchfunktionen zusammengezählt und verglichen. Anschließend wird eine einfaktorielle Varianzanalyse mithilfe von SPSS[1] durchgeführt. Die Analyse wird zeigen, ob die neu implementierte Suchfunktion signifikant mehr Hits generiert als die bestehende Confluence Suche.

[1] https://www.ibm.com/de-de/spss

6.3. Psychometrische Vorgehensweisen

6.3.1. ISO/IEC 9126

Die ISO/IEC 9126 legen Qualitätskriterien für die Entwicklung von Software fest. Es gibt sechs verschiedene Qualitätskriterien, welche sich wiederum in verschiedene Facetten einteilen lassen. Doch nicht alle der sechs Qualitätskriterien sind in dieser Arbeit von Relevanz. So werden Die Qualitätskriterien Wartbarkeit, Effizienz, Übertragbarkeit und Zuverlässigkeit nicht beachtet. Die Wartbarkeit bezieht sich auf die Fähigkeit von Software sich zu verändern. Diese Fähigkeit hat allerdings nichts mit der Qualität einer Suchfunktion zu tun, wie sie hier gemessen werden soll. Die Effizienz ist im Kontext von Suchfunktionen als gegeben anzunehmen. Der Author unterstellt hier, dass jede Suchfunktion, welche untersucht werden soll, auch effizient ist. Denn im Jahr 2023 ist anzunehmen, dass eine Suchfunktion auch schnell die Suchergebnisse darstellen kann. Die Übertragbarkeit beschreibt die Fähigkeit einer Software, auf verschiedenen Umgebungen lauffähig zu sein. Die Zuverlässigkeit beschreibt, ob die Software auch bei unterschiedlichen Rahmenbedingungen in der gewünschten Form funktioniert. Die beiden Kriterien liegen ebenfalls außerhalb des Scopes der Arbeit, weil auch sie sich nicht auf die Qualität der Suchfunktion als solches beziehen.

Die zu betrachtenden Qualitätskriterien sind Benutzbarkeit und Funktionalität. Die beiden Qualitätskriterien werden in den nächsten Kapiteln im Detail erläutert. Das schließt die verschiedenen Facetten der Qualitätskriterien mit ein.

6.3.2. Benutzbarkeit

Die Benutzbarkeit von Software teilt sich in die Facetten Attraktivität, Konformität, Erlernbarkeit, Verständlichkeit und Bedienbarkeit ein. Die Attraktivität soll hier nicht weiter betrachtet werden, weil die reine Optik einer Suchfunktion im Kontext dieser Arbeit keine Relevanz besitzt. Die Facetten Erlernbarkeit und Verständlichkeit ergeben sich durch die Konformität der Suche. Damit eine Suchfunktion konform ist, muss sie alle gängigen Arten von Suchen unterstützen. Also Keyword Search, Phrase Search, Boolean Search. Die unterschiedlichen Arten von Suchen wurden in Kapitel 3.4 näher erläutert.

Die übrige Facette ist die Bedienbarkeit. Eine Autovervollständigung von Wörtern kann es dem Softwareentwickler vereinfachen, passende Sucheingaben zu machen. Sie verhindert zum einen, dass er Tippfehler macht. Zum anderen hilft sie dem Softwareentwickler auch bei der Wortwahl. Sie sorgt also für eine bessere Bedienbarkeit.

6.3.3. Funktionalität

Die Funktionalität von Software teilt sich in die Facetten Angemessenheit, Sicherheit, Interoperabilität, Konformität, Ordnungsmäßigkeit und Richtigkeit auf.

6.3.4. Einfaktorielle Varianzanalyse

Die Varianzanalyse überprüft, ob ein signifikant unterschiedlicher Wert zwischen den beiden Suchfunktionen besteht. Wenn die neu implementierte Suchfunktion signifikant mehr Dokumente findet als die Confluence-Suche, dann bestätigt dies die Hypothese. Die Hypothese ist dabei, dass die neu implementierte Suchfunktion eine Verbesserung zur bestehenden Confluence-Suche darstellt.

Zur Durchführung der einfaktoriellen Varianzanalyse muss die abhängige Variable und die unabhängige Variable definiert werden. Die unabhängige Variable ist die verwendete Suchfunktion. Diese muss numerisch repräsentiert werden. So steht die Zahl 0 für die Confluence-Suche, und die Zahl 1 für die neu implementierte Suchfunktion. Die abhängige Variable ist die Anzahl der gefundenen erwarteten Ergebnisse. Sie ist die abhängige Variable, weil die Anzahl von der verwendeten Suchfunktion abhängt.

6.4. Wahl der Evaluationsmethoden

TODO:

7. Vergleich der Suchfunktionen

Zum Vergleich der beiden Suchfunktionen wird im Folgenden ein fiktionales Szenario dargestellt. Das Szenario beschreibt den Ablauf des Onboardings eines neuen Mitarbeiters. Dabei werden die expliziten Fragen beschrieben, welche sich der Mitarbeiter stellt, um sich einzuarbeiten. Es wird angenommen, dass der Mitarbeiter sich mithilfe der Wissensdatenbank einarbeitet und die Suchfunktion der Wissensdatenbank verwendet. Die Fragen, welche sich der Mitarbeiter des fiktionalen Szenarios stellt, werden anschließend bei der Durchführung einer Studie verwendet. Die Studie verwendet die Fragen, um die Performance zwischen Suchfunktionen zu vergleichen. Der genaue Aufbau der Studie ist im Folgenden näher beschrieben. Nachdem der Aufbau der Studie erklärt wurde, werden die Daten der Studie ausgewertet und dargestellt. Das Ergebnis wird diskutiert. Zuletzt wird auf die Validität der Daten eingegangen und es wird der Versuchsaufbau diskutiert.

7.1. Aufbau der Studie

Im Rahmen der Bachelorarbeit wird eine Studie durchgeführt, welche die neue Suchfunktion mit der bestehenden Confluence-Suche vergleicht. Da der Zeitrahmen begrenzt ist, in welchem die neue Suchfunktion entwickelt wird, wird nicht die tatsächliche Suchfunktion von Confluence entwickelt. Es wurde bereits erwähnt, dass die Suche von Confluence auf Apache Lucene basiert, und dass dieses ein VSM auf Basis von BM25 verwendet. Daher wird die neu entwickelte Suchfunktion in mehreren Konfigurationen verglichen. So wird die neue Suchfunktion so konfiguriert, dass diese eine reine BM25 Suche durchführt. Diese Konfiguration soll als Benchmark dienen und die tatsächliche Confluence-Suche repräsentieren. Eine weitere Konfiguration verwendet eine Mischung aus einer BM25 Suche, und einer semantischen Suche. Die beiden Suchalgorithmen werden zu gleichen Teilen verwendet. Zuletzt verwendet eine andere Konfiguration lediglich die semantische Suche auf Grundlage von Sentence Similarity.

Die Eigenschaft, welche untersucht werden soll ist, ob eine semantische Suche für den gegebenen Datensatz, und im Kontext einer Wissensdatenbank in der Softwareentwicklung, ebenfalls bessere Suchergebnisse liefert, als eine Suche auf Basis von BM25. Durch die Verwendung einer einheitlichen Implementierung wird sichergestellt, dass lediglich die Unterschiede der Suchalgorithmen untersucht werden. Es wird damit verhindert, dass die Confluence-Suche besser abschneidet, weil sie ausgereifter ist. Es wird ebenfalls sichergestellt, dass die Datensätze der Suchfunktionen identisch sind.

Für die Studie wurde ein Datensatz generiert, welcher Dokumente beinhaltet,

welche durch die Suchfunktion gefunden werden sollen. Der Datensatz wurde aus einem realen Softwareentwicklungs-Projekt generiert, indem ein Teil der tatsächlichen Confluence-Seiten exportiert wurde. Für jedes Dokument sind eine oder mehrere Sucheingaben definiert, mit dessen Eingabe das Dokument gefunden werden soll. Darüber hinaus ist für jedes Dokument festgehalten, zu welchem Anwendungsfall sich dieses zuordnen lässt. Die folgende Tabelle zeigt einen Ausschnitt aus den Daten, welche durch die Studie generiert wurde. Der Name des Projektes, aus welchem die Daten entnommen wurden, wurde unkenntlich gemacht.

Anwendungsfall	Bereich	Sucheingabe	Erwartetes Dokument
Onboarding	Entwicklung	Getting Started	Getting Started

Gefundene Dokumente	Hit	Hit in Five	Hit in Three	Hit in One
getting started (legacy)..., onboarding usecase x..., onboarding re..., ui-debian-build-base..., ep...	true	true	true	true

Die Suchfunktion gibt für jeden Algorithmus fünf Dokumente als Antwort auf eine Sucheingabe zurück. Diese fünf Dokumente werden nach dessen Score sortiert. Das bedeutet, dass das erste Dokument der Liste jenes ist, welches von dem Algorithmus als das passendste erachtet wird. Die fünf Dokumente werden darauf untersucht, ob sich das gewünschte Dokument unter den Dokumente befindet. Das Ergebnis ist ein Precision-Score für den Suchalgorithmus. Um eine detailliertere Analyse zu ermöglichen wird nicht nur die Precision in Bezug auf die ersten fünf Dokumente gemessen. Es wird die Precision für das erste Dokument (Hit in One), die ersten drei Dokumente (Hit in Three) und alle fünf Dokumente (Hit in Five / Hit) gemessen. Anschließend werden die Precision-Scores der Algorithmen miteinander verglichen. Dazu wird die Summe der Hits für alle Sucheingaben gebildet. Die Summe wird durch die gesamte Anzahl der Sucheingaben dividiert.

Die Studie wird vollkommen automatisch durchgeführt. Dadurch können Ergebnisse der Studie nicht durch Teilnehmer verfälscht werden. Es bedeutet auch, dass die Studie eine hohe Reliabilität hat und mit jeder Durchführung das gleiche Ergebnis liefert. Sie ist also reproduzierbar. Der Nachteil dieser Herangehensweise ist, dass die subjektive Wahrnehmung des Nutzers, in Bezug auf die Precision der Suchfunktion, nicht beachtet werden kann. So ist es denkbar, dass eine Sucheingabe nicht das gewünschte Dokument beinhaltet, aber andere Dokumente, welche ein Nutzer als sinnvoll erachten würde. Die Ergebnisse der Studie sind damit abhängig von der Vorauswahl der Dokumente, welche gefunden werden sollen, und der Sucheingaben, welche a priori als sinnvoll bestimmt wurden. Es ist möglich, dass eine Suchfunktion für eine Sucheingabe durchaus ein sinnvolles Ergebnis liefert, aber nicht das Ergebnis, welches durch den Aufbau der Studie erwartet wird.

7.2. Auswertung der Ergebnisse

Die Studie wurde mehrfach durchgeführt. Die Precision-Scores werden auf zwei Nachkommastellen gerundet. Die Precision-Scores werden in der nachfolgenden

Tabelle zusammengefasst.

Precision	Hit in Five	Hit in Three	Hit in One
BM25	0,56	0,5	0,39
Hybrid	0,56	0,52	0,35
Semantic	0,15	0,13	0,07

Es ist auffällig, dass der Precision-Score der semantischen Suche wesentlich schlechter ist als der Precision-Score der BM25 Suche. Dies könnte sich wie folgt erklären lassen. Bei der Implementierung der Suche wurden alle H1-Header, H2-Header und Paragraphen eines Dokuments zusammengefasst in jeweils einen String für das entsprechende HTML-Tag. Der Algorithmus der Suchfunktion vergleicht nun den String der Sucheingabe mit den Strings in den Dokumenten. Also mit dem String des Titels, dem String der H1-Header, dem String der H2-Header und dem String der Paragraphen. Weil die einzelnen Inhalte der HTML-Tags zusammengefasst werden, können diese Strings größer werden als die Sucheingabe, welche in der Regel zwei bis drei Wörter umfasst. Dadurch, dass die Sätze einen großen Größenunterschied haben, fällt entsprechend auch der Score der Dokumente niedrig aus. Da nun alle Scores niedrig sind kann die Suchfunktion nicht so leicht das passendste Dokumente finden. Dadurch werden zum Teil weniger relevante Dokumente gefunden und der Precision-Score fällt besonders niedrig aus.

Auch das Postprocessing könnte eine Auswirkung auf den Precision-Score der semantischen Suche haben. Für Bag of Words Modelle, wie BM25, ist es typisch ein Stemming vor der Indizierung durchzuführen. Zum einen hat dies positive Auswirkungen auf den Precision-Score. Zum sorgt dies dafür, dass verschiedene Tokens als das gleiche Wort betrachtet werden, auch wenn sie nicht in der gleichen morphologischen Form stehen. Die Details zu dieser Problematik wurden bereits in Kapitel 3.2 erläutert. Für die semantische Suche könnte ein Stemming allerdings negative Auswirkungen haben. Denn bei dem Training eines Sentence Transformers werden ganze Sätze betrachtet und nicht nur die Stammformen der einzelnen Wörter. Das bedeutet, dass der Sentence Transformer die Wörter nicht mehr korrekt semantisch zuordnen kann. Dies wirkt sich wiederum negativ auf den Precision-Score der semantischen Suche aus.

Die Hypothese ist also, dass der Precision-Score der semantischen Suche nur aus dem Grund gering ist, weil neben dem Title-Tag auch die weiteren Tags indiziert werden. Um diese Hypothese zu überprüfen wurde die Studie ein weiteres Mal durchgeführt. Dieses Mal wurden lediglich die Title-Tags indiziert. Das Title-Tag ist für jedes Dokument nur einmal vorhanden. Außerdem sind die Titelsätze der Dokumente wesentlich kürzer als ganze Paragraphen. Mit dieser Konfiguration sollte der Größenunterschied des Sucheingabe-Strings und der Titel-Strings kein entscheidender Faktor mehr sein. Das Ergebnis dieser Konfiguration war folgendes:**??**

In dieser Konfiguration hat die semantische Suche eine bessere Precision. Die Hit in Five Precision liegt bei 0,28 für die Konfiguration, welche nur den Titel

Precision	Hit in Five	Hit in Three	Hit in One
BM25	0,54	0,5	0,39
Hybrid	0,54	0,48	0,35
Semantic	0,28	0,22	0,11

Tabelle 7.1.: **Precision aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa**

der Dokumente indiziert, gegenüber 0,15 bei einer vollständigen Indizierung. Die Verbesserung der Precision kann mit drei Faktoren begründet werden. Zum einen die beiden Hypothesen, welche mit dieser Beobachtung untersucht werden sollten. Also zum einen der Größenunterschied von Sucheingabe-String und den Strings, welche indiziert werden. Zum anderen die Tatsache, dass sich das Postprocessing der Dokumente negativ auf die Fähigkeit des Sentence Transformers auswirkt, die Semantik der Wörter zu verstehen. Außerdem besteht die Möglichkeit, dass die Inhalte der Dokumente leicht von den Titeln der Dokumente abweichen können. Das würde bedeuten, dass für den gegebenen Versuchsaufbau der Titel besser geeignet ist, um die gewünschten Dokumente zu finden, als das gesamte Dokument. Die Precision für die BM25-Suche ist bei vollständig indizierten Dokumenten besser als bei alleiniger Beachtung des Titels. Lediglich die semantische Suche hat sich bei der zweiten Konfiguration verbessert. Das bedeutet, dass sich der gezeigte Effekt der Verbesserung der Suchfunktion tatsächlich mit der Hypothese begründen lässt, dass die Länge der Strings ein zu berücksichtigender Faktor ist. Da die semantische Suche weiterhin schlechter ist als die BM25-Suche, scheint es aber weitere Faktoren zu geben, welche zu berücksichtigen sind.

Eine mögliche Erklärung für die weiterhin schlechten Ergebnisse der semantischen Suche ist die Tatsache, dass es sich bei dem verwendeten Datensatz um eine Closed-Domain handelt. Das bedeutet, dass der Textcorpus Fachbegriffe beinhaltet, welche der Transformer mit hoher Wahrscheinlichkeit nicht kennt. Dementsprechend schwer fällt es dem Transformer folglich, passende Vektoren zu generieren, welche die Semantik von den domänenspezifischen Fachbegriffen abbilden. Die Paper *GPL: Generative Pseudo Labeling for Unsupervised Domain Adaptation of Dense Retrieval* (Wang, Thakur et al. 2022 von Wang et. al. und *TSDAE: Using Transformer-based Sequential Denoising Auto-Encoderfor Unsupervised Sentence Embedding Learning*Wang, Reimers et al. 2021) von Wang et. al. untersuchen Möglichkeiten, um Sentence Transformer an eine Domäne anzupassen. Dieser Vorgang wird als Domain Adaption bezeichnet.

7.3. Diskussion des Studienaufbaus

Wie bereits beschrieben gibt der Recall an, wie viele der relevanten Dokumente gefunden wurden. Die Precision gibt lediglich an, wie viele der Dokumente, welche gefunden wurden, relevant sind. Eine optimale Suchfunktion würde per Definition alle Dokumente finden, welche relevant sind, und keine irrelevanten Dokumente. Umgekehrt ist eine schlechte Suchfunktion eine Suchfunktion, welche keine rele-

vanten Dokumente findet, sondern nur irrelevante. Dabei spielt es für den Nutzer aber keine Rolle, ob irrelevante Dokumente gefunden wurden. Die Tatsache, dass die relevanten Dokumente nicht gefunden wurden, machen die Suchfunktion für den Nutzer nicht benutzbar. Die Studie untersucht die Precision der Suchalgorithmen. Auf Grundlage der obigen Argumentation zeigt sich, dass der Recall-Score wichtiger ist als der Precision-Score. Denn ist der Recall gering, dann ist die Suchfunktion nicht benutzbar. Eine Suchfunktion mit einem hohen Recall, aber eine niedrigen Precision könnte dagegen viele relevanten Dokumente finden, aber auch viele irrelevante Dokumente. Solange die relevanten Dokumente in der Liste weiter oben dargestellt werden, ist diese Zusammensetzung aus Precision und Recall gut.

Die Studie hat die Precision für das erste, die ersten drei und alle fünf Dokumente gemessen. Die Studie hat allerdings keinen Recall gemessen, obwohl es sinnvoll ist den Recall als den wichtigeren Score einer Suchfunktion zu erachten. Grund dafür ist die Tatsache, dass um den Recall messen zu können, alle relevanten Dokumente für eine Sucheingabe bekannt sein müssen. Um für eine Studie a priori Sucheingaben zu bestimmen, und alle Dokumente, welche für diese Sucheingabe relevant sind, müsste der gesamte Datensatz bekannt sein. Da der Datensatz mehrere hundert Dokumente beinhaltet ist dies nicht möglich. Und auch wenn der gesamte Datensatz bekannt wäre, dann müsste trotzdem eine Entscheidung darüber getroffen werden, welche Dokumente relevant sind und welche nicht. Das wäre wiederum eine subjektive Entscheidung, sodass die Validität des Ergebnisses anzweifelbar wäre.

8. Zusammenfassung und Ausblick

8.1. Zusammenfassung

8.2. Ausblick

9. Literaturverzeichnis

Antoniol, Canfora, Casazza und De Lucia (2000). „Information retrieval models for recovering traceability links between code and Documentation“. In: *Proceedings International Conference on Software Maintenance ICSM-94*.

Bar-Ilan, Judit (2002). „Methods for measuring search engine performance over time“. In: *Journal of the American Society for Information Science and Technology* 53.4, S. 308–319.

Castillo, Carlos (2005). „Effective web crawling“. In: *ACM SIGIR Forum* 39.1, S. 55–56.

Choudhary, Sneha, Haritha Guttikonda, Dibyendu Roy Chowdhury und Gerard P. Learmonth (2020). „Document retrieval using deep learning“. In: *2020 Systems and Information Engineering Design Symposium (SIEDS)*.

Clarke, Sarah J. und Peter Willett (1997). „Estimating the recall performance of web search engines“. In: *Aslib Proceedings* 49.7, S. 184–189.

Dengel, Andreas (2012). *Semantische Technologien Grundlagen - Konzepte - Anwendungen*. Spektrum, Akad. Verl.

Dit, Bogdan, Meghan Revelle, Malcom Gethers und Denys Poshyvanyk (2011). „Feature location in source code: A taxonomy and survey“. In: *Journal of Software: Evolution and Process* 25.1, S. 53–95.

Haiduc, Sonia, Gabriele Bavota, Andrian Marcus, Rocco Oliveto, Andrea De Lucia und Tim Menzies (2013). „Automatic query reformulations for text retrieval in software engineering“. In: *2013 35th International Conference on Software Engineering (ICSE)*.

Karpukhin, Vladimir, Barlas Oguz, Sewon Min, Patrick Lewis, Ledell Wu, Sergey Edunov, Danqi Chen und Wen-tau Yih (2020). „Dense passage retrieval for open-domain question answering“. In: *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*.

Khder, Moaiad (2021). „Web scraping or web crawling: State of Art, Techniques, approaches and application“. In: *International Journal of Advances in Soft Computing and its Applications* 13.3, S. 145–168.

Lehmann, Fritz (o. D.). *Semantic networks in Artificial Intelligence*. Pergamon Pr.

Li, Y., D. McLean, Z.A. Bandar, J.D. O'Shea und K. Crockett (2006). „Sentence similarity based on semantic nets and corpus statistics“. In: *IEEE Transactions on Knowledge and Data Engineering* 18.8, S. 1138–1150.

Manning, Christopher D., Prabhakar Raghavan und Hinrich Schütze (2019). „Term frequency and weighting“. In: *Introduction to information retrieval*. Cambridge University Press, S. 117–120.

Pugh, William (1990). *A Skip List Cookbook*. Techn. Ber. USA.

Reimers, Nils und Iryna Gurevych (2019). „Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks“. In: *Proceedings of the 2019 Conference on*

Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP).

Sarica, Serhad und Jianxi Luo (2021). „Stopwords in technical language processing“. In: *PLOS ONE* 16.8.

Sirotkin, Pavel (2012). *On Search Engine Evaluation Metrics.*

Treude, Christoph, Mathieu Sicard, Marc Klocke und Martin Robillard (2015). „TaskNav: Task-based navigation of software documentation“. In: *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering.*

Wang, Kexin, Nils Reimers und Iryna Gurevych (2021). „TSDAE: Using Transformer-based Sequential Denoising Auto-Encoderfor Unsupervised Sentence Embedding Learning“. In: *Findings of the Association for Computational Linguistics: EMNLP 2021.*

Wang, Kexin, Nandan Thakur, Nils Reimers und Iryna Gurevych (2022). „GPL: Generative Pseudo Labeling for Unsupervised Domain Adaptation of Dense Retrieval“. In: *Proceedings of the 2022 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies.*

Ye, Xin, Hui Shen, Xiao Ma, Razvan Bunescu und Chang Liu (2016). „From word embeddings to document similarities for improved information retrieval in software engineering“. In: *Proceedings of the 38th International Conference on Software Engineering.*

Zhang, Qin, Shangsi Chen, Dongkuan Xu, Qingqing Cao, Xiaojun Chen, Trevor Cohn und Meng Fang (2023). „A survey for Efficient Open Domain Question Answering“. In: *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers).*

Anhang

A. Anhang

A.1. docker-compose.yml File für Weaviate

```
version: '3.4'
services:
  weaviate:
    image: semitechnologies/weaviate:1.18.3
    command:
      - --host
      - 0.0.0.0
      - --port
      - '2000'
      - --scheme
      - http
    ports:
      - "2000:2000"
    restart: on-failure:0
    environment:
      PROMETHEUS_MONITORING_ENABLED: 'true'
      QUERY_DEFAULTS_LIMIT: 20
      AUTHENTICATION_ANONYMOUS_ACCESS_ENABLED: 'true'
      PERSISTENCE_DATA_PATH: "/var/lib/weaviate"
      DEFAULT_VECTORIZER_MODULE: text2vec-transformers
      ENABLE_MODULES: text2vec-transformers,qna-transformers
      TRANSFORMERS_INFERENCE_API: http://t2v-transformers:8080
      QNA_INFERENCE_API: "http://qna-transformers:8080"
      CLUSTER_HOSTNAME: 'node1'
    volumes:
      - /var/weaviate:/var/lib/weaviate
  t2v-transformers:
    image: semitechnologies/transformers-inference:sentence-transformers
    environment:
      ENABLE_CUDA: 0
  qna-transformers:
    image: electra-qna
    environment:
      ENABLE_CUDA: 0
```

A.2. Initialisieren des Schemas in Weaviate

```
WeaviateClass.builder()
  .className(DOCUMENT_CLASS)
  .properties(
    dataServiceHelper.buildProperties(
      mapOf(
        DOCUMENT_URL to WEAVIATE_TEXT_DATATYPE,
        TITLE_TAG to WEAVIATE_TEXT_DATATYPE,
        H1_TAG to WEAVIATE_TEXT_DATATYPE,
        H2_TAG to WEAVIATE_TEXT_DATATYPE,
        PARAGRAPH_TAG to WEAVIATE_TEXT_DATATYPE
      )
    )
  ).vectorIndexConfig(
    VectorIndexConfig.builder()
      .distance("l2-squared")
      .ef(100)
      .efConstruction(128)
      .build()
  )
  .invertedIndexConfig(
    InvertedIndexConfig.builder()
      .bm25(
        BM25Config.builder()
          .b(.5f)
          .k1(.5f)
          .build()
      )
      .build()
  )
  .vectorizer(VECTORIZER)
  .build()
```

Eidesstattliche Erklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt habe. Ich habe alle Stellen, die ich aus den Quellen wörtlich oder inhaltlich entnommen habe, als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Essen, den 23. September 2023