Aalto University
School of Science
Master's Programme in Computer, Communication and Information Sciences

Elmeri Niemelä

A Solution Retrieval Engine for a Customer-Facing Software Project Management System

Master's Thesis
Espoo, June 30, 2022

Supervisors: Professor Petteri Kaski, Aalto University
Advisor: Roy Nurmi M.Sc. (Tech.)

Aalto University
School of Science
Master's Programme in Computer, Communication and Information Sciences

ABSTRACT OF MASTER'S THESIS

Author: Elmeri Niemelä

Title:
A Solution Retrieval Engine for a Customer-Facing Software Project Management System

Date: June 30, 2022 **Pages:** 75

Major: Computer Science **Code:** SCI3042

Supervisors: Professor Petteri Kaski

Advisor: Roy Nurmi M.Sc. (Tech.)

In a growing expert organization, the ability to reuse solutions from previous work is crucial for maintaining an efficient business. This thesis utilizes the field of information retrieval to implement a solution retrieval engine aimed at helping the employees of SprintIT in solving their tasks. SprintIT uses an Enterprise Resource Planning (ERP) system with project management and other modules, which have accumulated knowledge of previously developed solutions in a form of unstructured text. Since SprintIT is a software company, many tasks are bug reports that have a stack trace as part of the description, which can be used as a strong signal for detecting duplicate problems. Texts containing information on previously encountered problems are gathered from the EPR system to a single search index, used to generate recommendations based on a given task. The ability to make ad hoc keyword queries to the same search index is included. The recommendations and search results are ranked and sorted in descending order of relevance, using one of the three different ranking approaches: The *Vector Space Model*, *Okapi BM25* and *BERT*. The ability to evaluate the effectiveness of different ranking methods and parameters is provided as part of the system. A selection of previously encountered problems are annotated with relevance judgements and used to make decisions on the best ranking method and parameters for this case study. Based on the evaluation, Okapi BM25 was most effective with a recall rate of 76% on a selection of test cases from the SprintIT ERP. As a result of this thesis, the ERP system of SprintIT gained information retrieval capabilities that allow SprintIT employees to utilize existing solutions to recurring problems.

Keywords: information retrieval, natural language processing, duplicate bug report detection, text ranking

Language: English

Aalto-yliopisto
Perustieteiden korkeakoulu
Tieto-, tietoliikenne- ja informaatiotekniikan maisteriohjelma

DIPLOMITYÖN TIIVISTELMÄ

Tekijä:	Elmeri Niemelä		
Työn nimi:	Ratkaisun hakumoottori asiakaskeskeiseen projektinhallinta järjestelmään.		
Päivyys:	30. kesäkuuta 2022	**Sivumäärä:**	75
Pääaine:	Tietotekniikka	**Koodi:**	SCI3042
Valvojat:	Professori Petteri Kaski		
Ohjaaja:	Diplomi-insinööri Roy Nurmi		

Kasvavassa asiantuntijaorganisaatiossa aikaisempien ratkaisujen uudelleenkäyttö on tärkeää tehokkaan liiketoiminnan ylläpitämiseksi. Tämä diplomityö hyödyntää aikaisempaa tiedonhaun tutkimusta hakumoottorin toteuttamisessa, jonka tavoitteena on auttaa SprintIT:n työntekijöitä ratkaisemaan heidän työtehtäviään. SprintIT käyttää toiminnanohjausjärjestelmää, johon kuuluu projektinhallinta sekä muita moduuleita, jotka sisältävät paljon kertynyttä tietoa eri asiantuntijoiden kehittämistä ratkaisuista. SprintIT on ohjelmistoalan yritys, joten työtehtävillä on usein virheilmoituksiin liittyvää suoritustietoa, jotka ovat vahvoja signaaleja aiempien kaksoisvirheilmoituksen tunnistamiseen. SprintIT:n toiminnanohjausjärjestelmän tekstikentistä kerätään dataa keskitettyyn hakuindeksiin, jota hyödynnetään työtehtävillä automaattisten ehdotusten luomiseen. Lisäksi toteutus mahdollistaa ad hoc-avainsana kyselyt samaan hakuindeksiin. Hakutulokset esitetään käyttäjälle laskevassa relevanssijärjestyksessä käyttäen yhtä kolmesta tunnetusta relevanssin arviointimenetelmästä: *Vector Space Model*, *Okapi BM25*, ja *BERT*. Järjestelmään kuuluu testipenkki, jolla pystytään arvioimaan näiden metodien sekä erinäisten parametrien vaikutusta haun tarkkuuteen ja tehokkuuteen. Diplomityö toteutetaan SprintIT:n omistamalla datalla, joten ennen haun arviointia testitapauksien hakutuloksia merkitään relevantiksi manuaalisesti järjestelmään kuuluvalla relevanssipäätöstoiminnolla. Tehdyn arvioinnin perusteella Okapi BM25 tuotti parhaat hakutulokset palautusprosentilla 76%. Tämän diplomityön tuloksena SprintIT:n toiminnanohjausjärjestelmä sai suositus- ja tiedonhakujärjestelmän, jonka avulla työntekijät pystyvät löytämään valmiita ratkaisuja toistuviin ongelmiin.

Asiasanat:	tiedonhaku, luonnollisen kielen käsittely, kaksoisvirheilmoituksen havaitseminen, relevanssilajittelu
Kieli:	Englanti

Acknowledgements

I want to express my deepest gratitude to my instructor Roy Nurmi who got me very excited about this thesis topic. Throughout this thesis, Roy helped tremendously by brainstorming ideas and giving extremely valuable advice.

In addition to Roy, I could not have undertaken this endeavour without the invaluable guidance of my supervisor Petteri Kaski. No matter what problem I had, Petteri always guided me to the correct path toward the solution, while also providing encouragement and valuable academic advice.

I would also like to thank the whole organization of SprintIT for enabling and funding this thesis. Special thanks to Laura Salo, Nestori Törmä, Otso Nurmi, Petri Järvinen and Riitta Saikkonen for agreeing to be test users for the system and for giving great improvement ideas as well as feedback. I am also grateful to my fellow thesis co-workers Miika Rouvinen and Samuel Johansson at SprintIT who provided a great environment for discussions and gave valuable tips and ideas for writing and improving the thesis. Many thanks to Matti Immonen as well for discussions, improvement ideas and for reading through the thesis and commenting on my writing. I'd also like to acknowledge my team leader Konsta Aavaranta who helped me to prioritize my efforts and my colleague Joakim Weckam who took over a large part of my previous responsibilities when I was occupied with my thesis. I'd also like to acknowledge Vitor R. Carvalho for sharing with me the source code of his email signature extraction program.

Thanks should also go to my family for all the encouragement and moral support. Special thanks to Essi Heikkinen, who supported me closely throughout this work and was always willing to help me by discussing problems and especially by proofreading my writing.

Espoo, June 30, 2022

Elmeri Niemelä

Contents

Chapter 1

Introduction

1.1 Motivation

Internal systems of a software company gather valuable domain-specific knowledge over time. Leveraging this knowledge has the potential to reduce time spent on resolving problems that have already been addressed in the past. In this case study, an Information Retrieval (IR) system is developed for a software company called SprintIT, in the confines of an Enterprise Resource Planning (ERP) system called Odoo. The goal of the IR system is to help employees find information from different parts of Odoo while working on a task.

A task refers to a single problem to be solved by an employee of SprintIT. All the work of SprintIT employees is organized into tasks making them an integral part of daily work. The tasks are most commonly related to either support projects, product development projects or customer implementation projects. A new task on a project of this type often raises questions such as:

1. "Has this issue been solved or discussed previously? What was the solution? What was the conclusion?"
2. "Do we already have a software module that implements something similar to this specification or need?"
3. "Who is an expert on this kind of problem?"

Without proper IR tools, finding the relevant information to these questions is time-consuming. A large amount of accumulated knowledge is in the form of unstructured text, scattered in different parts of Odoo, making it difficult to utilize.

Standard, non-customized Odoo has filtering tools that provide a way to narrow down a list of tasks or other relevant documents. However, using the

standard filtering tools, queries with multiple keywords are cumbersome to craft and require that you know the correct source field where the relevant text is expected to be found. In addition, filters can be crafted for only one document type at a time without the ability to search across multiple document types. The standard filtering system of Odoo is not designed for doing a system-wide search of the most relevant documents, however, it is currently the only available tool for finding previous solutions from the system.

Inadequate tooling causes multiple side effects that decrease the efficiency of the company. First, employees spend overlapping debugging efforts on problems that are already solved by others. Second, developers create duplicate software modules to address similar problems already solved in different customer projects. The third inefficiency is the lack of background context, usually missing from a task, which forces employees to spend considerable time looking for related information. In addition, the inability to easily find information causes many employees to forward their issues to the whole company chat channels in the hopes that someone would recognize the issue and give a solution. As the company grows in size, finding relevant information becomes increasingly harder making these inefficiencies increasingly significant. The implementation described in this thesis aims to provide relevant links to internal documents that aid in reducing the inefficiencies mentioned above.

1.2 Contribution and Structure

The approach of this thesis is to leverage proven methods of information retrieval to implement a content-based recommender system and an *ad hoc* search that aids in solving the software engineering-related tasks faced by SprintIT employees. The scope of the implementation is limited to applying three promising retrieval methods called the *Vector Space Model* [56], *Okapi BM25* [47, 48] and *BERT* [43] further introduced in Chapter 2. The implementation is also limited by the technology stack of the target environment into which the retrieval system is integrated. Section 2.7 discusses all the limitations that affected the scope of this thesis. The implementation utilizes well-established concepts of IR such as inverted indices, quantitative evaluation metrics and user relevance signals such as manually assigned relevance judgements and click-through data.

The closely related field of Duplicate Bug Report Detection (DBRD) has useful variations for information retrieval. For example, using the execution information i.e. the stack trace of a bug was shown by Wang et al. [66]

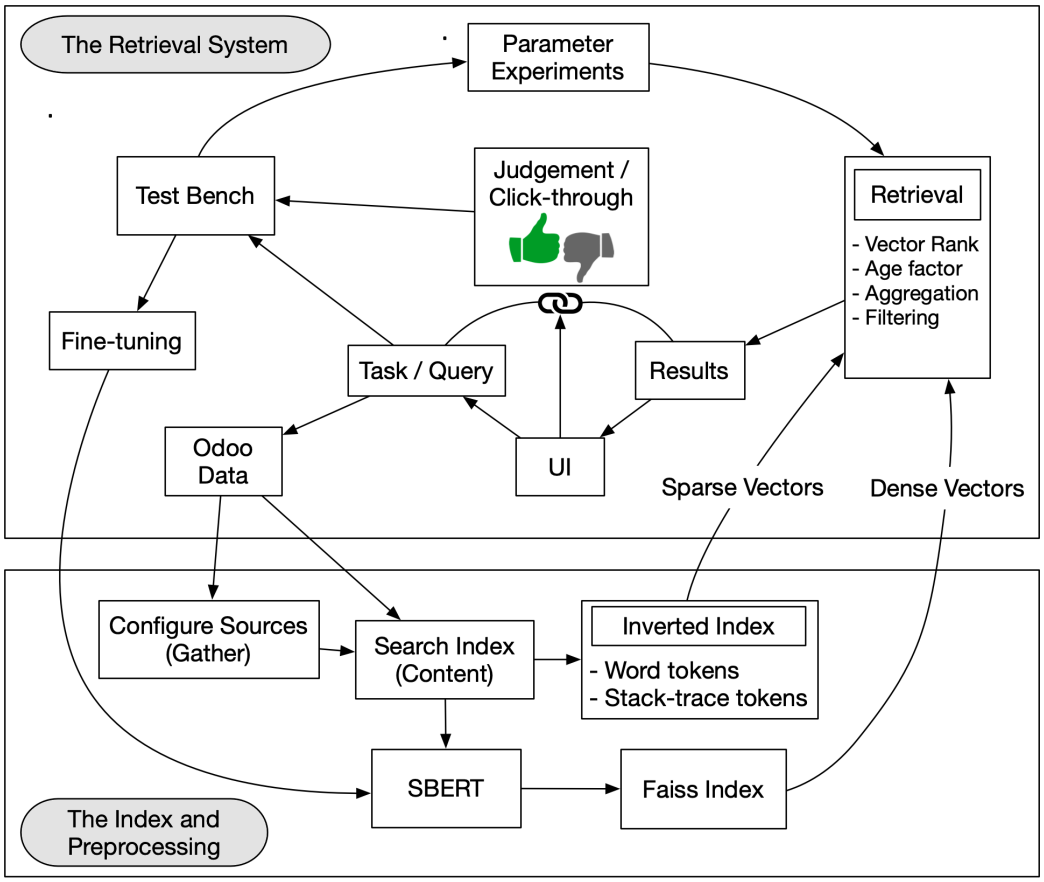

Figure 1.1: High-level overview of the system components. *The Index and Preprocessing* is the focus of Chapter 3 (lower half of the figure), and *The Retrieval System* is the focus of Chapter 4 (upper half of the figure). Even though the components interact with each other in many ways, the main information flow is represented by the arrows. The components are explained further in the following chapters and many of them are referenced throughout the thesis.

to increase recall, and the BM25 extension introduced by Sun et al. [59] improved recall when using long bug descriptions as a query. Following the previous research, this thesis utilizes both the stack trace information (see Section 3.4) and the BM25 extension for DBRD (see Section 2.4). Key contributions are listed below.

1. Performant IR engine built natively using the Odoo framework with Python/PostgreSQL backend.

2. Automatically generated recommendations based on the task submitted by an employee or a customer.
3. An ad hoc keyword search functionality for user-defined queries.
4. A dynamic and configurable search index that allows queries across all the relevant document types in Odoo. The configurable search index makes the implementation generic, allowing possible future use in customer implementation projects.
5. A Python stack trace tokenizer that distinguishes between error phrases, function names and lines of code.
6. An intuitive user interface that is guiding and effortless to navigate.
7. Simple annotation tool that enables storing user-assigned relevance judgements. Annotating retrieval results allows the usage of supervised learning methods in the future.
8. Automatic storage of user click-through data intended for providing relevance signales that can be used in more sophisticated ranking models in the future.
9. Proof-of-concept for retrieval with pre-trained language model BERT using the user-assigned relevance judgements for supervised fine-tuning.
10. An integrated test bench for a systematic evaluation and testing of different retrieval methods and hyperparameters.
11. Evaluation of the effectiveness with the different retrieval methods and hyperparameters.

The remainder of this thesis is organized as follows. Chapter 2 elaborates on all the necessary background information related to the implementation, first the case study specific background and then the utilized methodology from previous research. A large part of the implementation is the preprocessing and indexing of all the text scattered around different parts of the SprintIT Odoo. Chapter 3 describes the data, gathering it to a single index, preprocessing and related issues. Chapter 4 focuses on the retrieval side of the implementation, detailing the most important features of the system, design choices and the rationale behind those choices. With multiple retrieval methods and hyperparameters, a robust way of measuring the effectiveness of the retrieval system is crucial. Using a built-in test bench implementation, a systematic evaluation of all methods and justified choices for all hyperparameters are described in Chapter 5. Finally, Chapter 6 discusses ideas for future development and gives concluding remarks.

Chapter 2

Background

This chapter first addresses the background tied to the case-specific implementation and then discusses the theoretical background of the utilized methods and their applications in previous research. Section 2.1 elaborates on the case-specific information need and the inadequacy of existing search capabilities. Section 2.2 introduces the basic concepts of traditional term-based information retrieval. Sections 2.3 and 2.4 define two commonly used ranking functions for term-based IR: the *Vector Space Model* (VSM) and *Okapi BM25*. Section 2.5 goes beyond term-based models into neural network models, specifically a model called *BERT* and its derivatives for information retrieval. Section 2.6 provides a brief survey on applying the discussed models and other techniques in the closely related research field of duplicate bug report detection. Finally, Section 2.7 describes limitations that affected the implementation and approaches chosen in this thesis.

2.1 The Information Need

SprintIT[1] is a software company that helps customers in deploying and customizing ERP software called Odoo[2]. In addition to offering Odoo to its customers, SprintIT uses Odoo to manage its own business. Odoo is an open-source ERP software, that has a project management application along with other business applications as part of it. Odoo is modular and customizable, thus this thesis refers to "SprintIT Odoo" as the ERP system that has custom data structures and business logic related to the business needs of SprintIT. ERP is a central information repository with valuable knowledge of the core business processes of a company. In the case of SprintIT, knowledge

[1]https://www.sprintit.fi/
[2]https://www.odoo.com/

about previously developed solutions is written by the employees to the ERP software system on a daily basis. Effective utilization of this knowledge is a significant competitive advantage in the highly competitive ERP market.

Part of the SprintIT business model is to accumulate re-usable solutions to commonly occurring problems in ERP deployment. When faced with a problem from a customer, discovering if a solution to the same problem already exists is an everyday activity for the employees of SprintIT. Common problems include customizations to the standard Odoo applications, answering functional questions related to the usage of Odoo and resolving software bugs or data issues. When faced with such problems the prioritized information needs can be listed in the following order:

1. Solution to the same problem.
2. Recent discussion and work around the same problem area.
3. Name of an expert that has worked on a similar problem.

Finding the solution directly is the highest priority, but the recent discussion and names of related experts are often also useful. The standard filtering tools of Odoo are usually inadequate for satisfying any of the listed information needs.

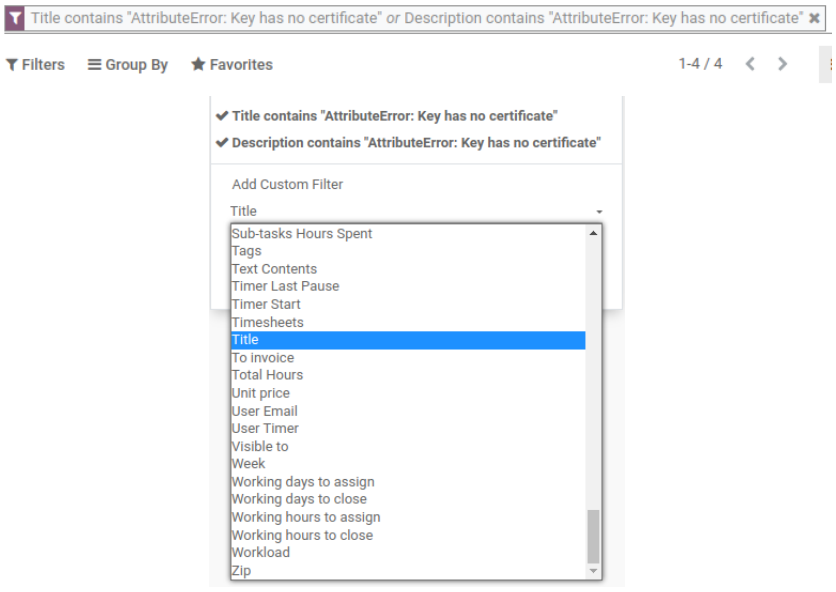

Figure 2.1: The user interface for constructing filters. Although the Odoo filtering system is flexible for limiting the contents of a specific view, it is not designed for system-wide information retrieval.

To highlight the inadequacy of the standard filtering tools of Odoo, let's consider an illustrative example case for an information need in a customer support project: The banking integration module raised an error to the customer where the last line of the stack trace is "AttributeError: Key has no certificate". To find previous solutions with the standard filtering tools we first have to select a substring to filter with and fields where we expect the substring might be found. Figure 2.1 illustrates selecting the error message as a substring to filter tasks on the fields title and description. It is worth noting that already the construction of such a simple filter takes anywhere from 30 seconds to 2 minutes depending on the filtering experience of the user. After the filter is constructed by the user, it goes through the Odoo framework and gets translated into an SQL search query over the two textual fields. The response time to the filter is 15 seconds on a database of 80 thousand tasks. If the user is lucky, the exact substring was indeed referenced previously on one of these two database fields. However, if the error phrase was previously mentioned only partially such as "Key has no certificate" it would remain undetected with this filtering approach. Furthermore, the title and description field of tasks are not even close to a comprehensive list of textual fields in the SprintIT Odoo that can resolve the information need. For example, the tasks have a chatter functionality attached to them which is a common place for finding relevant information. The database of SprintIT Odoo has 3.5 million chatter messages across thousands of different document types, which makes standard substring filtering even slower. In addition to the multiple relevant data sources already mentioned, the database of SprintIT Odoo has attachments, commit descriptions and module README files among other sources that are relevant when searching for solutions.

The example demonstrates that using the standard filtering system is not sufficient for finding solutions from SprintIT Odoo. Filtering one view of documents is not the same as providing the user with the most relevant documents from a variety of different sources. As the need for more sophisticated information retrieval tools is now established, Sections 2.2–2.6 review the literature for the existing methodology to address this need. Section 2.2 starts with the basic concepts of information retrieval.

2.2 Basics of Information Retrieval

Information retrieval is a study of storing, manipulating and searching information with a large collection of data. The data collection, also referred to as the corpus, contains a large set of text documents available for retrieval. The primary goal of IR is to provide users with the most relevant results

from the corpus, given the user's information needs. This is done by ranking the documents and displaying them in descending order of relevance.

The text ranking problem. As stated by Büttcher et al. [10] "The problem of scoring documents with respect to a query is one of the most fundamental in the field of information retrieval". Text ranking aims to provide an ordered list of relevant documents to a given query. The ordering is defined by a *ranking function* that takes as an input text *representations* of a query-candidate pair and returns a numeric score that describes how relevant the candidate is, compared to the query. To define the ranking function, comparable representations of the query and candidate document are needed. Based on these definitions, the text ranking problem can be divided into two subproblems:

1. **The representation problem**: representation of text that accurately captures the information content in a comparable way.
2. **The comparison problem**: a ranking function that accurately estimates the relevance of a text document representation with respect to a query.

This division into two subproblems follows the conceptualization of Lin et al. [32] re-phrased to a slightly more generic form.

The representation problem is traditionally addressed with a term-based "bag of tokens" approach, where each document is represented by the frequencies of tokens that comprise it. Tokens are character sequences that are usually words or as in this case study a combination of words and stack trace features. Tokens are extracted from a text document according to syntactic features such as punctuation or whitespaces. This is one example where information retrieval overlaps with the closely related field of *Natural Language Prosessing* (NLP). Tokenization is a fundamental design choice since it inherently limits the type of search queries supported by the system [10]. The implementation details of tokenization are discussed in Sections 3.2 and 3.4. Another approach for solving the representation problem is to develop a learned mapping from a text to a numeric vector using machine learning. A learned vector representation of a text is called an *embedding*. The BERT model, discussed in Section 2.5, is an example of a model that generates learned text embeddings.

A structure that links tokens to documents that they appear in is called an *inverted index*. The purpose of an inverted index is to efficiently prune a set of candidate documents into a smaller set. For example, if the query tokens are "hello" and "world" only documents that contain "hello" or "word" need to be considered for ranking and all other documents can be discarded

immediately. The inverted index is called *dynamic* when documents are continuously added, deleted or updated.

Multiple performance metrics have been developed to evaluate the performance of an information retrieval system. Büttcher et al. [10] divide IR performance metrics into two broad categories: efficiency and effectiveness. Efficiency is usually measured in response time (latency) or as throughput if the system has many simultaneous users. Effectiveness is more difficult to measure than efficiency, as it relies on human judgement to first annotate which results are relevant and which are not. The effectiveness measures considered in this thesis are *recall@k*, *nDCG@k* and *reciprocal rank@k*.

Recall measures the fraction of retrieved relevant results from all relevant results:

$$recall@k = \frac{N_{recalled@k}}{N_{total}}.$$

In this formula, $N_{recalled@k}$ is the number of relevant results recalled in the top-k results and N_{total} is the total amount of relevant results for the given query. Discounted Cumulative Gain (DCG) is a measure of ranking quality that takes into account the positions of retrieved relevant results:

$$DCG@k = \sum_{j \leq k} \frac{1}{\log_2(1+j)}.$$

In the *DCG@k* formula, j is a position of a relevant result in the retrieved list of results. The metric *nDCG@k* is the normalized version of *DCG@k* that is defined as

$$nDCG@k = \frac{\text{actual } DCG@k}{\text{ideal } DCG@k}.$$

Here, the ideal *DCG@k* is computed as if all the relevant documents would have been presented first in the list of retrieved documents. Finally, reciprocal rank measures how far down from the top the first relevant result can be found. Reciprocal rank is defined as

$$reciprocal\ rank@k = \frac{1}{i \in \{1, \ldots, k\}},$$

where i is the position of the first relevant recommendation in the resulting ordered list. Taking the average is a common way to aggregate these metrics across multiple test cases.

All the defined effectiveness metrics rely on relevance judgements made by a user. As relevance is inherently subjective, defining it precisely is difficult. An example of a subjective definition is that a text is only relevant

if it satisfies the information need of the person looking at it. For example, are apples relevant to oranges? Lin et al. [32] use this illustrative example for pointing out that relevance depends on the perspective of the user. From one perspective apples are not relevant to oranges, hence the common idiom "comparing apples to oranges", but if the user is looking for other fruits, then apples are relevant to oranges. The subjectiveness of relevance judgements affects the evaluation results of Chapter 5, where retrieval methods are compared based on metrics that rely on the relevance judgements mostly made by the author.

The excellent book by Büttcher et al. [10] is a recommended reading for a more comprehensive review of the fundamental concepts and techniques related to implementing IR systems. The book goes in-depth about all the basic techniques used in this thesis, including inverted indices, tokenization, probabilistic relevance, evaluation, vector space model and many other related topics beyond the scope of this thesis.

For the comparison problem, defined earlier in this section, multiple different ranking formulas exist. Sections 2.3 and 2.4 define two of the most commonly used methods to rank documents based on the bag of tokens document representation.

2.3 Vector Space Model

In 1975, Salton et al. [56] proposed a widely used text ranking model called the *Vector Space Model* (VSM). In the VSM, the bag of tokens representation is interpreted as a vector and the ranking is based on the angle between the query and the candidate vectors.

A candidate document D, as well as a query Q, are represented by T-dimensional sparse vectors, one dimension for each term in the vocabulary. Each vector component at position $t \in \{0, \dots T-1\}$ contains the term frequency i.e. the number of times a specific term occurs in a document. The VSM uses an additional weighting scheme called *Term Frequency - Inverse Document Frequency* (TF-IDF) which is applied to raw term frequencies. The TF component of TF-IDF is a logarithmically normalized term frequency. Other normalization schemes have been developed, but this thesis follows the original VSM paper [56] by using logarithmic normalization. The IDF component, developed by Jones [26], captures term specificity i.e. how specific the term is to the given issue. Intuitively, matches of very specific terms should weigh more when considering relevance. The IDF for a specific term

is defined as

$$\text{IDF}(t) = \log\left(\frac{N - df(t) + 0.5}{df(t) + 0.5} + 1\right),$$

where $df(t)$ is the number of documents where the term t appears and N is the total number of documents in the corpus. The TF-IDF weighting scheme is defined as follows:

$$\text{TF-IDF}(D_t) = \log(1 + D_t) \cdot \text{IDF}(t).$$

In this definition, D_t is the frequency of term t in document D. After applying a weighting scheme such as TF-IDF, to both the query document Q and a candidate document D a common similarity measure called *cosine similarity* is computed as

$$\text{cosine similarity}(Q, D) = \frac{\sum_{t=0}^{T-1} D_t Q_t}{\sqrt{\sum_{t=0}^{T-1} D_t^2}\sqrt{\sum_{t=0}^{T-1} Q_t^2}}, \tag{2.1}$$

where T is the dimensionality of the vectors, in this case the size of the vocabulary. Note that cosine similarity is a generic similarity measure between two vectors, thus it can also be used for different vector-based document representations such as BERT embeddings discussed later in Section 2.5.

Even though the VSM is simple and intuitive, it has been criticised for its heuristic nature. The euclidean length normalization in the cosine similarity denominator has proven to be inadequate in datasets with a large variation in document lengths [10]. The BM25 method described in Section 2.4 has a better theoretical justification and has proven to be more effective in many real-world IR implementations.

2.4 Okapi BM25

Okapi BM25 (BM25) is one of the most successful ranking methods in information retrieval [32]. Developed initially for the Okapi system in the 1990s by Robertson et al. [47, 48], it has become a baseline method in IR research. In addition, BM25 is used as the default ranking method on many commercial IR systems [32] such as Apache Lucence [5]. Practically, BM25 is quite similar to VSM as it performs an inner product of sparse term vectors, but it is derived from the *Probabilistic Relevance Framework* (PRF) with the following assumption:

> If retrieved documents are ordered by decreasing probability of relevance on the data available, then the systems effectiveness is the best that can be obtained for the data.

This is commonly known as the *Probability Ranking Principle* [45], re-phrased to a shorter from by Jones, Robertson et al. in [27, 49]. It describes the perspective from which BM25 was derived from. A commonly used formulation of BM25 is defined as follows:

$$\text{BM25}(Q, D) = \sum_{t \in Q \cap D} \frac{D_t}{D_t + k_1 \cdot (1 - b + b \cdot \frac{l_D}{L})} \cdot \text{IDF}(t).$$

In this formula, the sum is over all the common terms t in Q and D, D_t is the raw frequency of term t in document D, l_D is the length of document D i.e. the total word count and L is the average length of documents in the corpus. The parameter k_1 controls term saturation i.e. the diminishing returns of higher occurance frequency of a specific term in the candidate document D. The parameter b controls the importance of document length normalization. Robertson et al. [49] suggest based on a significant number of experiments that using values $0.5 \leq b \leq 0.8$ and $1.2 \leq k_1 \leq 2$ works well in most circumstances.

Some publications also add an extra $k_1 + 1$ multiplier that removes the effect of term saturation when term frequency is 1, however, it can be omitted from practical implementations as it does not change the relative order of ranked candidates. For a more in-depth description of the theoretical framework behind BM25, the book by Robertson et al. [49] is recommended.

Robertson et al. [50] introduced a simple extension to BM25 called *BM25F* which allows utilizing the structure of the documents. The idea is to construct the query vector Q and the candidate vector D from a collection of fields $f \in F$ each associated with a weight $w(f)$. Doing the weighted combination of multiple source fields at the representation level instead of combining at the final score level retains the desirable properties of BM25 such as term saturation and length normalization. In this thesis BM25F is defined in a limited sense: the text source field has a weight associated with it however the frequencies in D or Q are not combined from multiple fields as there is no document structure generally defined. The documents in SprintIT Odoo are asymmetric and retrieved from different sources. Thus in this thesis, BM25F is defined as

$$BM25F(Q, D) = BM25(w(f_Q)Q, w(f_D)D).$$

In this definition, f_Q and f_D are the source fields for Q and D respectfully from which the term frequency vectors are extracted. Note that $w(f_Q) = w(f_D) = 1$ falls back to standard $BM25$.

An extended version of BM25F called $BM25F_{ext}$ was designed by Sun et al. [59] to address longer queries typical in duplicate bug detection. It adds an extra factor based on query term frequencies to the $BM25F$ formula:

$$BM25F_{ext}(Q, D) = \text{BM25F}(Q, D) \cdot \sum_{t \in Q \cap D} \frac{(k_3 + 1)Q_t}{k_3 + Q_t}.$$

The parameter k_3 controls the contribution of query term frequencies to the score. When $k_3 = 0$ or $Q_t = 1$ it has no effect on the score, but if $k_3 > 0$ and $Q_t > 1$, it emphasises the contribution of term t to the resulting score. The initial BM25 formula also had a similar parameter called k_3 but it was omitted later as it had little effect for small queries. Throughout this thesis, the implemented method BM25 refers to $BM25F_{ext}$ and the standard BM25 is implicitly evaluated when $k_3 = 0$ and field weights are all 1.

VSM and BM25 both interpret the bag of tokens representation as a term frequency vector in which the full dimensionality is the number of unique terms in the vocabulary. The dataset of this thesis contains approximately 320 000 unique tokens while the average document has 26 unique tokens. This makes the vectors incredibly sparse, filled with mostly zeroes. The term *sparse vector method* is used to classify methods that utilize this type of document representation. In contrast to sparse vector methods, the term *dense vector method* refers to methods using representations of the text where the dimensionality is orders of magnitude smaller and the values are mostly non-zero. The BERT model introduced in Section 2.5 produces dense vector representations of text documents.

As the basis for both the VSM and BM25 is the exact matching of terms between the query and candidate documents, they both perform poorly when the same underlying topic is described in different terms. This is commonly referred to as the *vocabulary mismatch problem* and approaches to alleviate this problem are discussed in Chapter 6. Furthermore, the bag of tokens approach ignores the surrounding context of words, as no positional information is encoded in the word frequencies. To alleviate the issues of term-based representations, a vast amount of recent research has focused on producing more sophisticated text representations. Section 2.5 introduces one of the more advanced text representation models called *BERT*.

2.5 BERT for Information Retrieval

In recent years, neural network-based language models have gained popularity over the traditional "bag of tokens" representations in text ranking.

From the plethora of existing models, this section focuses on promising IR applications of a generic language model called *BERT*. For a broader survey, Guo et al. [18] provide a recent comparative study on multiple different neural ranking models. BERT is based on the *Transformer* architecture which offered a parallelized alternative to the sequential computation of recurrent neural networks. For more details on Transformers, see the original paper by Vasvani et al. [64] and the *Annotated Transformer*[3] guide by Rush et al.

BERT which stands for *Bidirectional Encoder Representations from Transformers* is a language representation model, pre-trained on unlabeled text and fine-tunable for a variety of downstream tasks [15]. It improves on previous models by utilizing a bidirectional pre-training objective that results in new state-of-the-art results on multiple NLP tasks. In addition to various language modelling tasks, BERT has seen success in information retrieval. Google announced the usage of BERT to better understand conversational search queries where prepositions like "for" and "to" affect the semantics[4], and Yang et al. [70] applied BERT to ad hoc document retrieval which demonstrated improved retrieval precision.

The idea behind the bidirectional pre-training objective is to mask a portion of the input sequence and use the surrounding context to predict the masked portion. This is referred to as the *Masked Language Model* (MLM) objective, which was shown by Delvin et al. [15] to produce high-quality token embeddings. The objective of predicting a masked portion of text conditions the model to utilize contextual information in the text. Diagnostics of BERT have shown evidence that this conditioning leads to, for example, semantic role recognition such as preferring "to tip a waiter" over "to tip a chef" [54]. The ability to encode contextual information into a text representation supersedes the capabilities of the bag of tokens representation model, which does not encode any positional information. After the unsupervised pre-training phase, BERT can be further fine-tuned on a relatively small amount of labelled domain-specific data.

The first application of BERT for information retrieval can be conceptualized as a *cross-encoder*: A query and a candidate *sentences* are tokenized into a single *sequence* that is fed as an input to the transformer network. The terminology is adopted from the original paper [15] where *sentence* refers to an arbitrary span of text and *sequence* refers to an ordered array of tokens. The input token sequence consists of WordPiece [68] tokens and other special tokens such as the classification token ([CLS]) that is inserted as the first token for every input sequence. The output is an array of contextualized to-

[3]http://nlp.seas.harvard.edu/annotated-transformer/

[4]https://blog.google/products/search/search-language-understanding-bert/

ken embeddings, including an embedding for the special classification token ([CLS]). The relevance score between the query and the candidate is typically produced by feeding the ([CLS]) token into a fully connected classification layer. The full input/output specification is explained in more detail in the original paper [15] and a simple example cross-encoder implementation using BERT for relevance ranking is described by Nogueira et al. [39].

The cross-encoder approach has a major scaling problem when applied to IR: the computationally costly BERT inference has to be made for every candidate in the corpus with respect to one query. Due to this reason, the most common way of applying the cross-encoder BERT is to only re-rank a small set of top results retrieved by traditional IR methods. To improve performance for retrieval, Reimers et al. [43] developed a bi-encoder architecture on top of the pre-trained BERT, called *Sentence-BERT* (SBERT). With this adaptation, retrieval time was drastically reduced, allowing BERT to be effectively used in end-to-end IR[5]. For example, finding the most similar pair from a collection of 10000 documents was reduced from 65 hours with BERT to 5 seconds with SBERT [43]. Instead of inferring the relevance score between a query-candidate pair directly, SBERT produces fixed-size document embeddings, pre-computable offline, which allows the usage of a simple similarity-metric without a major loss of accuracy. Thus at search time, only the query needs to be embedded using SBERT, and the rest of the computations are fast inner products to pre-computed dense vectors. BERT produces token embeddings for every token in the input sequence. To produce the fixed-sized embedding for the whole document, the individual token embeddings need to be combined by applying a pooling operation. In this thesis, the default pooling strategy of mean-pooling is used for SBERT, which simply takes an average of all the token embeddings produced by BERT. The relevance score is computed as the cosine similarity (Formula 2.1) between the query and candidate embeddings. The resulting score is scaled with a fixed constant to accommodate the *Multiple Negatives* loss function discussed later. Inferring the relevance score directly using a cross-encoder is slightly more accurate compared to SBERT, thus it is common to use a cross-encoder for further re-ranking the top results retrieved by a more efficient method. See Figure 2.2 for an illustration of the architectural difference between cross-encoders and bi-encoders.

In this thesis, the SBERT bi-encoder is evaluated as a potential retrieval method and the BERT cross-encoder is used as a re-ranker. The bi-encoder uses a *Multiple Negatives* [19] loss function during fine-tuning. When de-

[5]A Python library implementation of SBERT called *sentence-transformers* can be found from https://www.sbert.net/

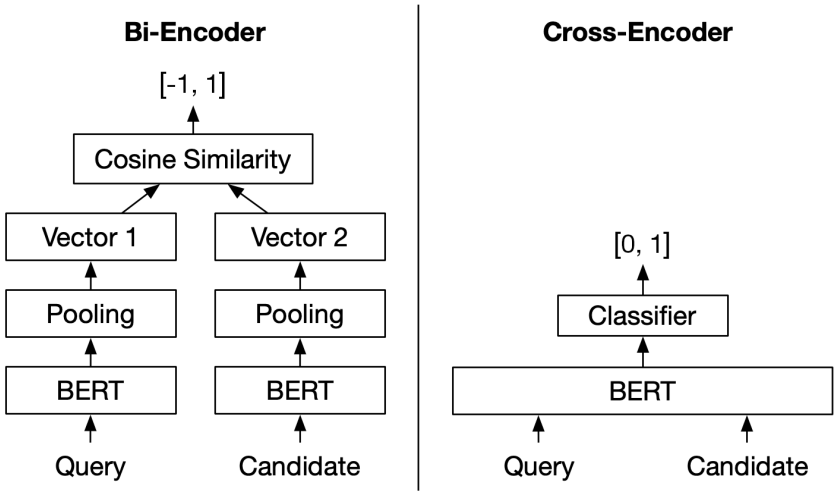

Figure 2.2: The architectural difference between bi- and cross-encoders, originally drawn by Reimers et al. [43]. Compared to a cross-encoder BERT which uses a classifier to directly infer the relevance score, the bi-encoder processes the candidate and query separately, using pooling operations to produce fixed-size sentence embeddings that can be compared using cosine similarity. The utility of bi-encoders is that they produce storable document embeddings that can be ranked without making BERT inferences. Cross-encoders are more accurate, but they make costly inferences at query time for each possible candidate, making them inefficient at a larger scale.

scribing the loss function, a *positive result* refers to a retrieval result that was judged to be relevant and a *negative result* refers to an unannotated, likely irrelevant retrieval result. A *hard-negative* refers to a candidate document whose embedding is similar to the query but is not annotated as relevant.

Each training batch of size N consists of triples (q_i, p_i, n_i) that are selected randomly from annotated training examples. In these triplets, q_i is the query text, p_i is a randomly selected positive result for the query and n_i is a randomly selected hard-negative for the query. As the name *Multiple Negatives* suggests, the loss function assumes that all the other $N-1$ candidates in the same batch are also negative examples irrelevant to q_i. This creates a dependence on batch size, causing larger batch sizes to generally yield better results. The multiple negatives loss function minimizes the negative log-likelihood for softmax normalized cosine similarity scores. The cross-encoder fine-tuning minimizes the binary cross-entropy loss between target and input logits having positive results labelled with a value of one and hard-negatives with zero.

This thesis utilizes the compressed version of BERT called MiniLM [65] for both the cross- and the bi-encoder. Normal BERT models consist of millions of parameters which make fine-tuning and inference latency a problem in many real-world applications. MiniLM addresses this by mimicking the self-attention modules of BERT using a smaller model. This process is called deep self-attention distillation.

A key problem in fine-tuning BERT for information retrieval is the selection of non-relevant documents. Karpukhin et al. [28] found that using in-batch negative sampling is the most effective approach. In this approach, a mix of random documents from other examples in the training batch and hard-negatives retrieved by BM25 is used for the non-relevant documents during training. The non-relevant results retrieved by BM25 are considered as hard-negatives since they have a high score with exact word matching but yet are non-relevant. Xiong et al. [69] argued that even better hard-negatives can be produced by using the same model being trained, compared to using BM25 which biases the model to mimic sparse retrieval.

Information retrieval requires a low latency response to a query. An inverted index, as discussed in Section 2.2, is trivial for term-based document representations, but can not be directly utilized when there is no explicit link between one vector component and a specific textual feature of the document. On the other hand, an exhaustive scan on a large set of dense high-dimensional vectors is quickly impractical even with an efficient ranking function such as cosine similarity. For this reason, using SBERT embeddings for IR relies on an efficient *Approximate Nearest Neighbour* (ANN) search. For a formal definition of the ANN problem, refer to Indyk et al. [22]. An efficient and popular open-source implementation for ANN problem called Faiss was developed by Facebook [25] in 2021. A Python library called *txtai* [35] provides simple abstractions for the process of indexing SBERT document embeddings to a Faiss index. This thesis utilizes a Faiss index via txtai in the implementation of the BERT-based retrieval method.

The research area of *Duplicate Bug Report Detection* (DBRD) is closely related to the goal of finding previous solutions to software-related problems of SprintIT customers. Section 2.6 provides a brief survey of previous research, applying the VSM, BM25 and BERT among other methods to DBRD.

2.6 Duplicate Bug Report Detection

The field of *Duplicate Bug Report Detection* (DBRD) aims to automatically detect multiple bug reports that describe the same underlying problem. The

purpose is usually to reduce the number of open bug reports, but in our case, we aim to utilize the information provided by previous bug reports to solve the customer's issue more rapidly.

Natural language processing has been used previously for DBRD. Runeson et al. [55] implemented a prototype for DBRD using logarithmic word frequencies in the vector space model. In a similar approach, Hiew [20] used TF-IDF weights instead of plain logarithmic word frequencies when calculating the textual similarity between bug reports. Hiew compared new bug reports to aggregate centroids of previously detected duplicate bug clusters instead of comparing individual bug reports directly.

In addition to using the natural language of the bug reports, Wang et al. [66] proposed a novel method of using *Execution Information* (EI) i.e. the function call stack associated with the bug. They used VSM with TF-IDF weights for both the natural language and EI components and chose the granularity for the EI tokens to be the canonical signature of each method in the call stack. The authors note that the function level granularity is not sufficient to always distinguish between different bugs since an implementation of one function may contain multiple unrelated bugs. To address this issue, they introduced credibility thresholds to ignore execution information over natural language information when the EI signal was not strong enough. The approach of function-level granularity differs from the approach of this thesis: in addition to function names, more distinct features such as lines of code and error phrases are extracted. The evaluation Chapter 5 discusses the experimental results with different stack trace features.

The approach of Jalbert et al. [24] and Sun et al. [60] was to use a discriminative model called *Support Vector Machines* (SVM) for DBRD. Given a pair of TF-IDF weighted vectors, an SVM classifier is trained to discriminate between duplicate and non-duplicate bug report pairs. Sun et al. [60] was able to outperform the approaches of [24, 55, 66] when only considering the natural language of bug reports.

More recently, BM25 has become a common text ranking method used for DBRD. Sun et al. [59] demonstrated that BM25-based method can outperform TF-IDF-based methods in DBRD. They developed a retrieval function called REP that used structured information such as product, component, priority etc. in addition to using textual similarity with word and bigram tokens. After the work of Sun et al. [59] many other papers have also used BM25 for DBRD [1, 3, 21, 37, 63] and for duplicate question detection in stackoverflow [2, 71].

A Bugzilla extension called NextBug [51] provides recommendations of open bugs to developers using the VSM. Rocha et al. [52] made an empirical comparison of REP [59] and NextBug in suggesting similar bugs and con-

cluded that while they are close in effectiveness, NextBug provided slightly better results. The authors made an email survey to developers in the Mozilla ecosystem from which 67% of (44 out of 66 developers) expressed interest in seeing recommendations for similar bugs in their bug reporting system.

Papers [23, 37, 71, 71] used topic modelling, specifically *Latent Dirichlet allocation* (LDA) as part of the similarity score. For more details on applying LDA to text modelling, refer to the paper by Blei et al. [7]. The approach of Nguyen et al. [37] was to use the BM25F score combined with the similarity of topic distributions modelled with LDA. Using a topic modelling as part of the similarity is a suggested branch of possible future research, further discussed in Section 6.2.

Since the release of BERT and SBERT in 2019, a couple of publications have applied BERT-based architectures to DBRD. Isotani et al. [23] used finetuned SBERT with a triplet loss for DBRD and achieved higher accuracy compared to baseline systems using TF-IDF and LDA. Similarly to this thesis, they evaluated the model using a proprietary dataset, therefore it is not known how their model compares to approaches evaluated on open datasets. Rocha et al. [53] took a clustering approach to DBRD using BERT for contextual features and LDA for topic modelling. They reported increased recall rates on common datasets compared to previous state-of-the-art machine learning approaches [8, 9] and [14]. However, the comparisons are not perfect due to the unknown holdout split used by the previous authors. The authors of [8, 9, 14, 53] all used the same datasets Eclipse, NetBeans and Open Office generated by Lazar et al. [31].

Before discussing the implementation side, it is important to recognize the limiting factors of this thesis. Section 2.7 discusses the technical limits and why some existing IR system implementations were not utilized as a part of the implementation.

2.7 Limitations

Odoo ERP has a specific technology stack: PostgreSQL relational database management system, a Python backend for Odoo processes and a web-based JavaScript frontend. From an architectural perspective, a simple implementation that can utilize the existing technology stack without introducing new integrations or programming languages is preferable. Furthermore, the current execution environment called *Odoo.sh* is a shared cloud platform for Odoo ERP that has its own limitations:

1. Wall-clock processing time is limited to 900 seconds per Odoo process.

2. Random-access memory available for the retrieval system is limited to approximately one gigabyte.
3. Installation of external libraries is limited to packages on *The Python Package Index* (PyPI)[6]. For example, adding new PostgreSQL extensions or using the package manager of the underlying operating system is not allowed.

These limitations can be circumvented by moving parts of the system to another server, but doing so is not desirable as it would break the standard deployment process of Odoo addons making future usage in other projects more difficult. An extra server adds the need for over-the-network data synchronization, which would require more maintenance effort compared to a built-in addon. As the memory-hungry BERT and Faiss combination does not meet these strict requirements, the BERT-based retrieval is built as a proof-of-concept and left out of scope for the system deployed to production use. If the execution environment is changed from Odoo.sh to a less restrictive environment in the future, this proof-of-concept can become a viable retrieval method for production use.

Text-based information retrieval is a well-studied field, with multiple open-source tools available. One of the best known enterprise IR implementation is called *Elasticsearch* [17], which is based on *Apache Lucence* [5]. Using Elasticsearch as a backend for this case study has notable downsides. First, Elasticsearch is implemented in Java, which does not fit the existing technology stack of Odoo and SprintIT. Integrating a large application with an unfamiliar codebase would require extra efforts during the implementation and it would make maintenance more costly as well. The second problem is that deploying the Elasticsearch backend to production would require a separate server due to the technical limitations of Odoo.sh discussed earlier. Again, the maintenance effort would increase as the searchable data would need to be synchronized to another server over the network. For these reasons, Elasticsearch was left out of the scope of this thesis. However, Elasticsearch does have a rich set of scalable features, therefore it is a suggested path for future research.

One further limitation is the time budget allocated to this thesis. All the development and writing related to this thesis is to be done within 6 months as agreed upon with SprintIT. Due to this reason, multiple promising improvements to the system are left for future research and discussed in Chapter 6.

The final production system uses only the existing technology stack of Odoo which includes PostgreSQL and Python and a limited set of additional

[6]https://pypi.org/

libraries. This is not a major concern, as Python has a lot of tooling for natural language processing and PostgreSQL is a performant database system for storing and retrieving data. With these limitations in mind, Chapter 3 shifts the focus from the background to the implementation side by describing the searchable index and the preprocessing steps required for effective retrieval.

Chapter 3

The Index and Preprocessing

The contents of the searchable index and the associated preprocessing lay the foundation for effective information retrieval. Preprocessing refers to transformations applied to a text that make it suitable for retrieval. Preprocessing includes text cleaning i.e. the process of removing irrelevant signals from the text that would produce inadmissible results if not removed. The more irrelevant signals are left in the index after text cleaning, the harder it becomes to produce relevant retrieval results. In addition to cleaning the text, preprocessing transforms the text into a representation that can facilitate information retrieval. This chapter explains what data sources are indexed for retrieval, how the text is processed and what are the biggest sources of irrelevant signals while giving illustrative examples. Section 3.1 starts by describing the process that gathers all the text sources into a single data model, its generality and SprintIT configuration. After the text is gathered, it is pushed through a text processing pipeline that extracts different features and removes as many irrelevant signals as possible. Section 3.2 explains the design of the text processing pipeline. Emails sent by SprintIT customers contain reply structures and signature phrases that interfere with producing relevant recommendations. Section 3.3 discusses the special cleaning procedure applied to emails. Another special case is the tokenization of Python stack traces. Section 3.4 focuses on how and why stack traces are processed separately from the rest of the text.

3.1 Configuring the Index

The implementation allows configuring source tables and fields to be indexed for retrieval. The configuration is designed such that the same system can be utilized for other customers with different information needs. The source

Table 3.1: Dataset overview. The first column shows the source field name, the second column has the document type to which the retrieval result link will point and the last column shows the approximate amount of documents in SprintIT Odoo. Many document types in Odoo have a message thread for discussions and notes. These messages contain problem descriptions received by email, specification discussions, debugging notes and problem solutions shared by employees. In support projects, attachments usually contain a screenshot or a text document with some error message received by the customer. Over 75% of the attachments are JPEG or PNG images, thus an OCR pytesseract library [58] is used for extracting the text from the images. README files and commit descriptions of modules originate from Bitbucket/Github integration and provide knowledge of existing solutions to problems. SprintIT uses Ansible automation for deploying module updates. In case an update-related problem has been encountered previously, the deployment error and the user who encountered the error are saved to the system which provides a link to a user who has encountered the problem. Document pages form a small internal wiki for documenting useful knowledge.

Source Field	Linked Document	Approximate Count
Message Body	Task	121 000
	Lead	10 000
	Module Template	2 000
	Ansible Server	500
	Document Page	200
Description	Commit	39 000
Attachments	Task	28 000
	Lead	1 000
	Module Template	100
	Document Page	100
Title	Task	28 000
Description		25 000
Technical Name	Module Template	10 000
Description	(git repository)	5 000
Deployment Error	Ansible Server	3500
Title	Document page	200
Content		100
		275 000

fields are defined through a new data model called *IR Gather* that collects the data from the configured sources into another data model called *IR Content*. The IR Content data model stores the content and metadata for the documents in the searchable index and encapsulates the logic related to indexation.

The chosen granularity for indexation is the contents of one source field on a single source record. Assume that $field\_id$ uniquely identifies a source field and row_id uniquely identifies a single source record. Then exactly one corresponding IR Content record is created to match the $(field\_id, row\_id)$ pair which provides a direct link to the source data. The IR Content data model stores the $(field\_id, row\_id)$ pair and the text gathered from the source. In addition, IR Content stores all the relevant metadata for displaying the related document as a retrieval result to the user. A retrieval result that a user interacts with is a link between two instances of IR Content: the query instance, that initiated the retrieval and the candidate instance, that was the result of the retrieval.

Note that this type of data model does not have a document structure defined. All instances of IR Content are thought of as separate even though they can be closely related to one document. For example, two distinct messages on the same task have the same $field\_id$ which is the *Body* field of Odoo's message data model, but they have distinct row_ids. Ranking methods such as the BM25F variant of BM25 (discussed in Section 2.4) can utilize a proper document structure but this is difficult to define as the documents from different sources (listed in Table 3.1) are asymmetric and do not follow one specific document structure. A configurable document structure is left for future research and discussed briefly in Section 6.2.

The data source configuration for this case study is described in table Table 3.1. In addition to task-related data such as titles, descriptions, discussions and attachments, the SprintIT Odoo contains multiple other useful sources of information. Most of the sources are in plain text or HTML, but attachments contain different datatypes including images that are converted into plain text. The logic of the IR Content data model i.e. text preprocessing, creating the document representation and storing it to the inverted index is discussed in Section 3.2.

3.2 Text Processing Pipeline

The IR Content index normalizes the differences between the original data sources. For example, some of the indexed text fields contain HTML code and some contain plain text. For tokenization, the HTML formatting

needs to be removed but the web interface that displays the results expects HTML. Therefore conversions from both directions, HTML-to-plaintext and plaintext-to-HTML, are needed to normalize the storage format for both use cases. Normalizing the indexed text simplifies further processing steps as the input can be expected to have a consistent format.

After the data is normalized, the text processing pipeline starts by cleaning the text partially based on its origin. In the case where the origin is the message table, text snippets containing email signature and reply patterns are extracted to a separate field, stored only for debugging purposes. The HTML is converted into plain text while preserving linebreaks. Preserving linebreaks is important for the stack trace tokenizer described Section 3.4. All HTML tags are removed with *Regular Expression* (regex) substitutions and the HTML entities are unescaped into plain text. The final preprocessing step is the removal of excessive whitespace, URLs and punctuations from the raw text. Different stages of the preprocessing pipeline are inspectable from the system settings UI which makes the preprocessing procedure transparent and easy to debug.

The natural language text is tokenized into words based on whitespace and punctuation characters. To normalize inflexions, words are stemmed using the algorithms developed by Porter, M. [41] in the Snowball language [42]. For the dataset of this case study, stemmers for both English and Finnish are needed since the suffix rules are different for each language. The Python library *nltk* [6] implements the Snowball stemmers for English and Finnish. To determine which stemmer to use, a language detection for each word is done with a library developed by Shuyo, Nakatani [57] and ported to Python by Michal Danilák [12]. This approach is not perfect, for example, the Finnish word "tilinumerot" was wrongly classified to be English, causing the stemmed word not to be the same as the stemmed word of "tilinumero". Failure to stem correctly reduces the possible set of matching documents and increases the specificity (IDF) of the word causing it to have a bigger than expected effect on the resulting relevance scores. To address this issue, a couple of additional techniques were experimented with: using Google services to translate Finnish into English as well as double stemming with both Finnish and English suffix patterns. Unfortunately, these techniques did not have a significant effect on recall rates and thus were discarded. Although common, word-level tokenization is not the only approach to natural language tokenization. General problems related to word tokenization and alternative approaches are discussed more in Section 6.1.

Common stopword lists provided by the Python *nltk* [6] library for both English and Finnish are used. In addition, the Finnish stopword list is extended with stopwords provided by Diaz [16]. Some additional words are

marked as stopwords including employee names and common words used in the company business context such as "odoo" or "module" that have little information value. Words with only numeric characters are also marked as stopwords. The system allows adding additional stopwords if needed, by toggling a field called "active" on the word list view. Stopwords are only discarded by the automated recommendations. The user-made ad hoc queries take the stopwords into consideration, thus stopwords are not discarded during indexation, only flagged as non-active. The ad hoc search is discussed more in Section 4.1.

During tokenization, the link between documents and tokens is stored in a database table that will act as an inverted index. For each token t occurring f times in a document d a triplet (t, d, f) is stored to this table. The usage of inverted indices is discussed further in Section 4.4.

3.3 Cleaning Emails

Emails often contain signature phrases and reply structures with previously sent messages appended to the email body. If left as part of the indexed document, the words in the signature and reply structures can easily skew the document representation away from representing the actual information contents, producing irrelevant search results. In some cases, the email client uses a delimiter such as two dashes "-" to mark the start of a signature block. Commonly known delimiters added by email clients combined with other recurring patterns are used for cleaning the irrelevant signals caused by emails. However, as seen from Figure 3.1 some email clients do not use a detectable pattern that could be reliably used to extract the signature phrase. The email clients do not have a common standard for marking the start of a signature block and some email clients append the signature phrase to the email without any delimiter. Company slogans and disclaimers, as seen in Figure 3.1, are commonly used as part of the signature causing irrelevant retrieval results, especially if used as part of the task recommendation query.

As an attempt to address these difficult cases, a crude statistical heuristic is used: if an identical sequence of words on the same line appears in more than ten messages, the line is disregarded. This simple approach seemed reasonably effective in removing repeating disclaimers that many of the SprintIT customers use as part of the email signature. However, this approach does not work for emails of new customers that have sent less than ten emails previously to SprintIT Odoo. Another possible drawback is that some lines with relevant information may get removed by mistake when they are repeated in ten different messages. However, at the time of writing the SprintIT database

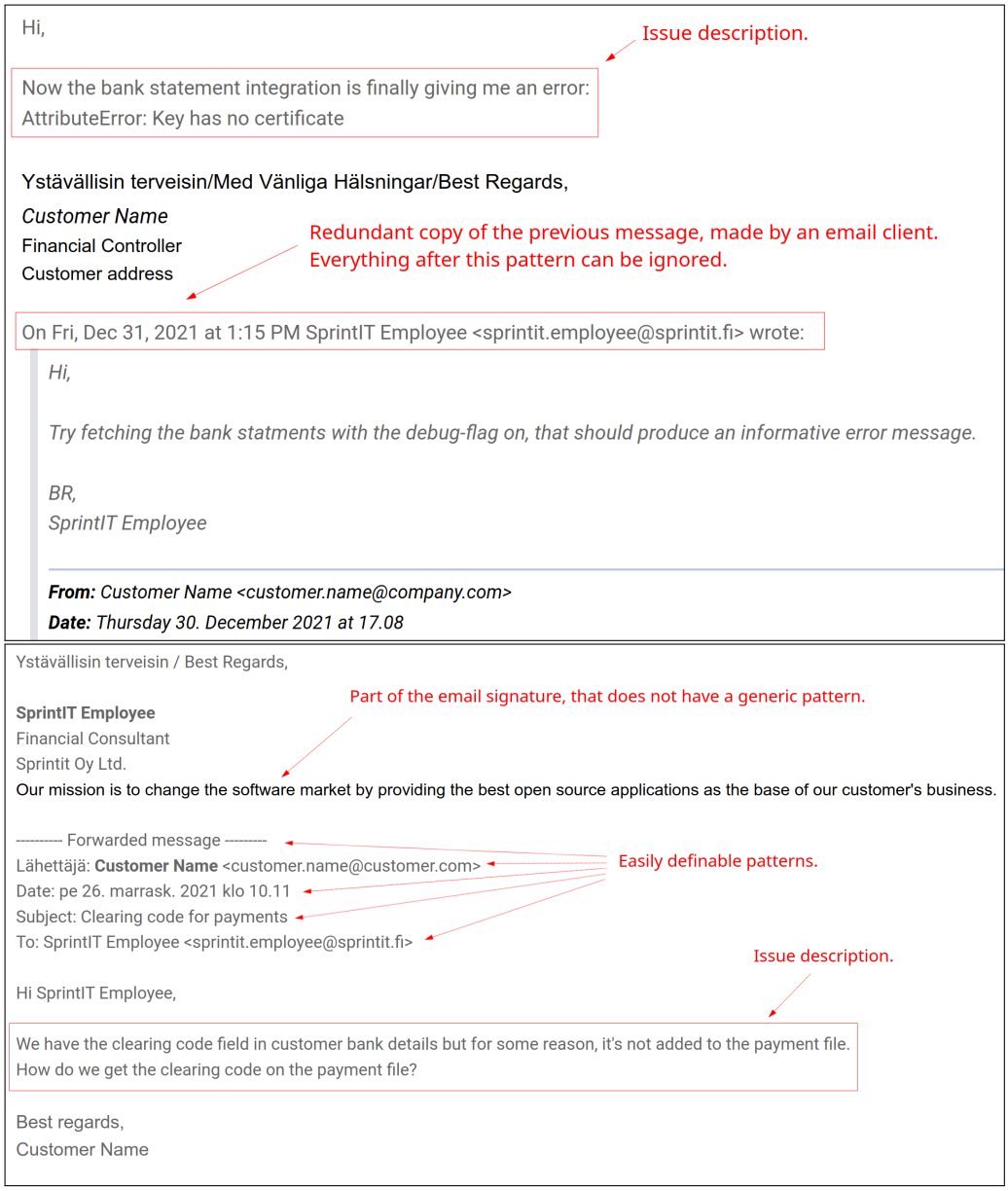

Figure 3.1: Two illustrative examples that highlight the irrelevant parts of email texts. Many email signatures do not have a general pattern that could be used for extraction. The email on the top is an example where the email client aggregates all the previous discussions into the new email. The email on the bottom is an example where an employee has forwarded a customer issue to our support email in order to generate a task in to the system.

has only approximately 700 lines that were repeated in more than ten messages and had a length of over 10 characters. By browsing through the repeated lines, it is clear that they are overwhelmingly irrelevant. Nonetheless, it is acknowledged that this heuristical apparatus is not an elegant solution and possible future improvements are discussed briefly in Chapter 6.

3.4 Stack Trace Tokenizer

For a software company, a piece of a stack trace can be a strong signal for detecting similar problems. However, a full stack trace contains many irrelevant parts related to the Odoo framework. Lower parts of the call stack are often identical for many different error conditions. To extract the relevant parts and disregard the irrelevant parts, a separate tokenization process is defined for stack traces.

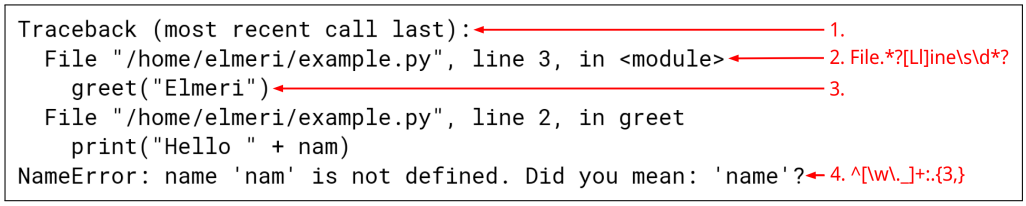

Figure 3.2: Stack trace tokenization. 1. Detect that text contains a Python stack trace. 2. Match a "File" pattern. 3. Save as a token: the line of code that comes right after the "File" pattern. 4. Save as a token: the error phrase that matches the error pattern and is 1-4 lines below the last seen "File" pattern.

Figure 3.2 illustrates the structure of a Python stack trace and how it is tokenized. The figure displays some features with distinctive patterns that can be utilized in tokenization. However, some key features such as the line of code (feature number 3 in Figure 3.2) only have a detectable pattern on the previous line. Due to this observation, the tokenization process is defined to rely on the linebreaks in the stack trace text. Thus, it is crucial that the preprocessing of text, described in Section 3.1, preserves new lines as otherwise there is no detectable pattern for extracting the lines of code and separating them from error phrases.

Table 3.2 enumerates the patterns used for different types of stack trace features. In addition to classifying stack trace features based on textual patterns, two line-based counters are used: *Depth* counter and *After-File* counter. Both counters are incremented by one per line of text processed.

The purpose of the Depth counter is to discount features that are at the bottom of the stack call stack. The discount is justified as the lower parts of a stack trace are the most specific for any given error. This specificity combined with token IDF, allows relevant errors to be separated from irrelevant stack traces. As the stack trace features are weighted highly on specificity, they do not displace as many relevant matches based on natural language similarity. Compared to the word tokenization process, the inverse of the Depth counter is used as a substitute for word occurrence frequency during ranking and it is stored similarly in the inverted index. Depth discount is calculated distinctly for all feature types and additionally, each feature type is associated with a tunable weight parameter. The Detect typed features are only used to reset the two counters and they are not utilized in retrieval whereas the Function, Error and Line tokens are saved as part of the document representation. The After-File counter is set to zero when a pattern starting with "File" is detected and its purpose is to aid in the detection of features that follow the "File" pattern. For example, the line of code feature is extracted only when the After-File counter equals one. If another exception occurs during the handling of the previous exception, the Python interpreter outputs two stack traces in which case the Depth counter is reset as Detect type pattern is detected.

Table 3.2: Typed feature patterns extracted by the stack trace parser and their relationship with the two line-based counters. The first column shows the regular expression used to detect the feature. The second column has the feature type and the last column describes the interaction with the line-based counter variables. Detect typed features reset the respective counter variable. Error and line-of-code type features use the after file counter for extraction.

Feature	**Type**	**Counter**
The above exception was the direct cause of the following exception:	Detect	Depth $\leftarrow 0$
During handling of the above exception, another exception occurred:	Detect	Depth $\leftarrow 0$
`Odoo Server Error`	Detect	Depth $\leftarrow 0$
`\(most recent call last\):`	Detect	Depth $\leftarrow 0$
`File.*?[Ll]ine[\s\d]*?`	Detect	After-File $\leftarrow 0$
`File.*?[Ll]ine[\s\d]*?,[\s]*in[\s]*([><\w]+)?`	Function	
`^[a-z_][a-z0-9_\.]+:.{3,}?`	Error	After-File $\in \{1, \ldots, 9\}$
`^Path:.{3,}`	Error	After-File $\in \{1, \ldots, 9\}$
`^Node:.{3,}`	Error	After-File $\in \{1, \ldots, 9\}$
`^xmlid:.{3,}`	Error	After-File $\in \{1, \ldots, 9\}$
`<any pattern>`	Line	After-File $= 1$

Figure 3.3 displays three example results of the described stack trace tokenization process. For the term-based methods, BM25 and VSM, the stack trace tokens are included in the bag of tokens document representation and

they are ranked identically to word token matches. For SBERT embeddings in the Faiss index, the extracted stack trace tokens with non-zero weight are appended to the input text that is fed through the SBERT transformer network. In addition to using stack trace tokens as part of the document representation, each stack trace token is indexed as a searchable document. The resulting stack trace token view can be used to browse the list of all documents that include the same piece of the stack trace.

Token	Content Count	Token Length	IDF ...	Info Type	Active
button_immediate_install	90	24	4,29	Function Name	☑
ValueError: Invalid field 'function' on model 'ir.cron'	2	55	7,88	Raised Error	☑
where_clause, where_params, tables = Rule.domain_get(self._name, mode)	13	70	6,20	Line of code	☑

Figure 3.3: Example results from stack trace tokenization. As seen from the first column, stack trace tokens are case sensitive and store punctuation marks as well to increase specificity. The content count column refers to the number of documents that reference the specific token. The IDF is computed based on the content count as described in Section 2.3. Info type column refers to the feature type of the token as listed in Table 3.2. The active field can be toggled by a system administrator to neglect the effect of specific tokens during the generation of recommendations.

Having explained the searchable index and how it is constructed, Chapter 4 focuses on how the index is utilized to provide relevant retrieval results.

Chapter 4

The Retrieval System

Once the text is indexed, we can turn our focus to the aspects of the system that use the index to retrieve and display relevant information. Section 4.1 uses an example support task to illustrate the user's possibility to utilize the two different retrieval processes i.e. task recommendations and ad hoc searches. Section 4.2 focuses on the results of the retrieval: how the results are displayed to the user and what type of user interaction is expected. Section 4.3 describes the different factors that determine the final rank of a candidate document and how the ranking methods introduced in Chapter 2 are applied in practice. Section 4.4 discusses optimizations on computational efficiency and illustrates an example pitfall that caused poor performance in the first versions of the system. Finally, the test bench implementation, that produced the quantitative evaluation results of Chapter 5, is outlined in Section 4.5.

4.1 Initiating Retrieval

A user can utilize the retrieval system in two ways: by browsing the automatically generated task recommendations or by manually formulating an ad hoc keyword query. We first focus on the task recommendations. As the goal of this thesis is to improve the efficiency of employees, the best way to achieve maximal efficiency is to automatically provide a solution to a task.

An ordered list of recommendations is displayed to the user as part of the task interface. Figure 4.1 shows an example support task with the title and description describing an issue faced by a customer. To generate recommendations based for the task, the source text for the recommendation query needs to be defined. The best query text on a task can be in one of many places such as title, description, chatter or attachments. However, using all

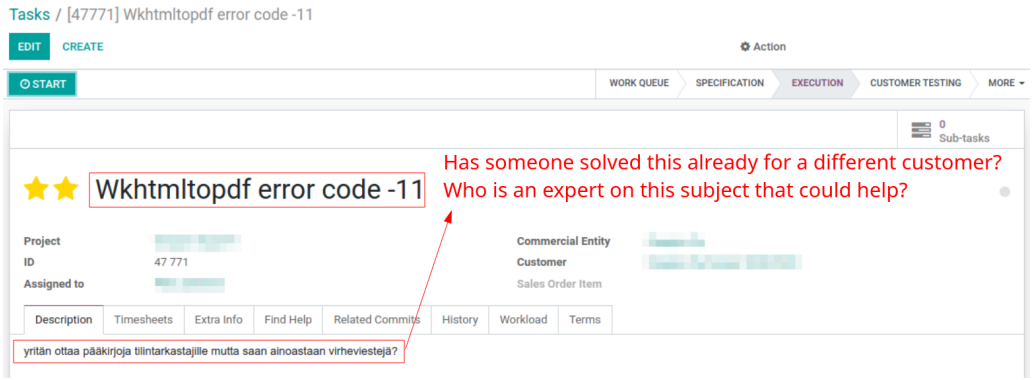

Figure 4.1: An example task with a customer-given problem. The task title and description, highlighted in the picture, are used as the query text for automatically generated task recommendations. The information need of the support team is illustrated by the two question phrases in the image.

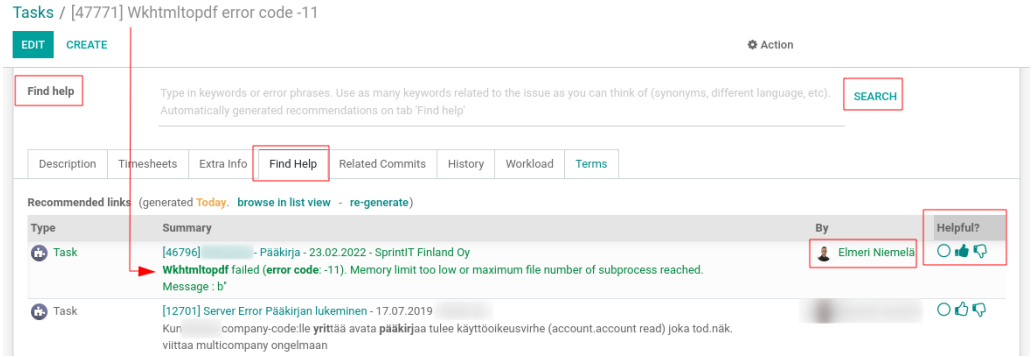

Figure 4.2: The solution recommender provides a recommendation view on the task, where the first recommended link satisfied the information need of the support team in this example case. In addition, the user can make an ad hoc keyword search directly from the same view.

of these sources as part of the query adds an excessive amount of irrelevant signals leading to inferior recommendations. Therefore, selecting the query text from a task is a compromise; using a small set of source fields can lead to missing the essence of some tasks while using a large query can result in irrelevant recommendations for other tasks. Most often the original idea of a task is captured by the title and description fields thus they are selected as the primary source for the query text. However, SprintIT Odoo allows for tasks to be created from an email sent by a customer, in which case the email body is saved as the first message on the task chatter and the description will be empty. Therefore, if the task description is empty, the title and the first message are used as the query text. Since the first message is an email, using

it increases the irrelevant signals in the query as discussed in Section 3.3. Nevertheless, the initial email from the customer usually contains the issue description thus it is worth using as part of the recommendation query.

The real-world example case in Figure 4.2 shows that the system can automatically find a solution for the given problem. However, if the recommendations do not address the information need of the user, the user can make an additional ad hoc keyword search. For example, when a customer describes two distinct issues in one email the automatically generated recommendations can be dominated by links only relevant to one of the issues. The ad hoc search uses the same retrieval pipeline as the automated recommendations with one key difference: stopwords are **not** excluded. This is justified by the assumption that when the user is directly formulating a set of keywords to search with, he or she knows which words are important for the query and which are not. If stopwords were excluded, the results for a query such as "server not working" would be ranked only by the words "server working" which could easily result in a confusing order of results. The results of ad hoc searches are displayed the same way as automated recommendations. Section 4.2 focuses on how the results are displayed to the user and the expected user interaction.

4.2 User Interaction

To boost user productivity, the user interface (UI) has to be intuitive, guiding and effortless to navigate. While Figures 4.1 and 4.2 already displayed the overall UI of a task, Figure 4.3 focuses more specifically on the UI design of the retrieval results.

The most important part of the retrieval result UI is the summary text that allows rapid skimming through the results to quickly identify links of interest. The summary is implemented by highlighting the highest IDF tokens that are common between the query and candidate text. The visualized document type helps the user to look for a specific type of recommendation. For example, if the task is describing a customization specification, then the user will be primarily looking for links with the type "Module".

In an expert organization, the user related to the recommended document can be the key to solving the problem. As seen from Figure 4.3, a user is associated with each resulting recommendation or search result. Even if a solution is not found directly, the results can contain similar problem descriptions in which case sending a message to the related user can be helpful. Clicking the picture on the user icon opens a direct personal message chat which makes contacting the related expert convenient.

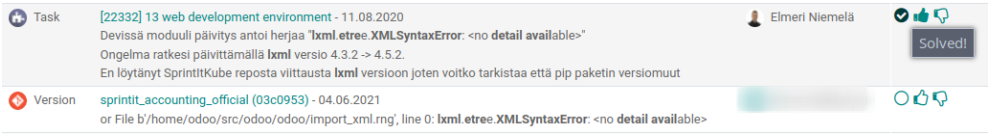

Figure 4.3: Visualizing results in the UI. The left-most column visually relates the document type behind the link to the application icon from which the document originates. In the centre is the document title with a direct link and creation date, followed by a summary text describing the link content. The next column has the related user of the document accompanied by a small picture of the user. Finally, the right-most column allows the user to make a relevance judgement or to quickly see if the result is already deemed relevant by another user.

The left-most column of Figure 4.3 with thumbs up and down buttons as well as the "Solved!" checkmark, allows the user to make a relevance judgement regarding the result. The main purpose of the user-provided relevance judgements is to evaluate and improve the ranking system. As discussed in Chapter 2, the relevance judgements are the basis for all the different metrics used for effectiveness evaluation. An effortless way of manually annotating relevant results during development was crucial since the development was done on a proprietary dataset that was initially unannotated. Furthermore, during production usage, annotating links provides a way for users to mark found information as useful, such that the findings can be easily reused later. As the user-assigned relevance judgements accumulate a larger annotated dataset over time, the accumulated data can be used in the development of more sophisticated ranking models. Ideas for future research regarding different ranking models are discussed briefly in Chapter 6.

In addition to relevance judgements, click-through records are also collected as user feedback to serve two purposes. First, they provide relevance signals automatically without requiring additional action from the user, which allows for improving the system even when the user does not manually annotate the results. The second purpose of the click-through data is to monitor how much the system is utilized by the SprintIT employees to assess the significance of the system. Storing the click-through records is implemented as an on-click mechanism inside the Odoo JavaScript framework. In addition, the links in the HTML body are changed from direct links to intermediary links that first store the click-through before redirecting the user. Changing the links covers the case where the user copies the link address, instead of directly clicking it with the left mouse button.

Internal links posted by users in the chatter or task description are parsed and stored using the same retrieval result data structure. When a task is referenced on another document, the reference is displayed above the automatically generated recommendations. Parsing internal links from messages allows users to make two-way links between documents that connect relevant information. In addition, the manual linking facilitates a mechanism for annotating false negatives in retrieval: if a relevant task is known but not recommended by the system, the user can link it manually and then click "thumbs up" to store the relevance judgement. This way the missing retrieval result is recorded in the system and automatically included in the test bench discussed further in Section 4.5.

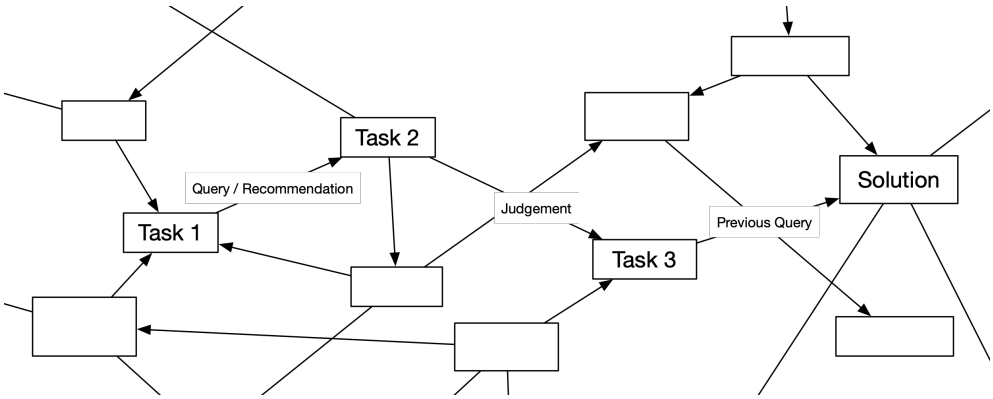

Figure 4.4: The links between related tasks are stored along with the user provided relevance signals. The information graph constructed from the links can be easily navigated from the user inteface, allowing users to jump from one related task to another. Furthermore, interpreting the relevance signals in a graph structure can facilitate building a graph-based ranking algorithm such as PageRank [40] in the future.

The relevance judgements, documents linked by users and click-through records gradually couple together related information. Relevance judgements allow the next user that is searching for similar information to immediately detect the links that are already marked as helpful. Most of the generated recommendations point to other tasks that also have recommendations. This allows a user to use the recommendations for jumping from one relevant task to another until they find a solution. In addition, all the ad hoc queries made from the task UI are listed in the same tab as recommendations, which allows other users to reuse previous search efforts. The annotated search results and recommendations can be viewed as an information graph, illustrated in

Figure 4.4, where the judgements are guiding the user's traversal through the graph. When searching for help, the user can jump from one task to another task guided by the relevance judgements.

4.3 Ranking Pipeline

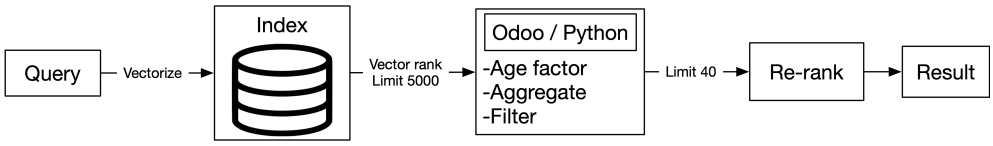

Figure 4.5: Different parts of the ranking pipeline, starting from the input query on the left-hand side and finally returning the most relevant results on the right-hand side.

Figure 4.5 illustrates the different stages of the ranking pipeline. First, the query is vectorized with the same procedure as the already indexed candidate documents. Then initial ranking based on the vectors is performed on the index layer that is PostgreSQL for the sparse vector methods and Faiss for the dense vectors produced by SBERT. The initial rank returned by PostgreSQL is a close approximation of the VSM or BM25 rank, and the rank returned by the Faiss index is the cosine similarity between the query and the candidate. To formulate an efficient SELECT statement on PostgreSQL, the index side ranking uses a value of one for the query token frequencies. Experimental evidence showed that using a sufficiently large limit between the index and application layer made this approximation of the real ranking method have no effect on the final recall.

After initial ranking, a limited set of candidate documents retrieved from the index are further processed on the Odoo/Python application side. For the VSM and BM25, the approximated ranking score, returned by the SQL SELECT, is recalculated on the application side to account for the token frequencies of the query. Then the score is multiplied with an age factor that discounts older documents:

$$\text{Age Factor}(x) = 1 - k + \frac{1}{ax^2 + \frac{1}{k}}. \tag{4.1}$$

In Formula 4.1, x is the age of the document in years measured from its creation date, the parameter k controls the maximum discount factor when x approaches infinity and a controls the slope. The discount is non-linear

to achieve two properties. First, the age factor should matter less when documents are within a year of age. Second, extremely old documents should not be completely discarded. Figure 4.6 illustrates how the score gets affected with increased age, using the parameters $k = 0.4, a = 0.05$. Discounting the age is justified by the rapid development cycle of Odoo software that has major updates every year, which rapidly reduces the relevance of older solutions.

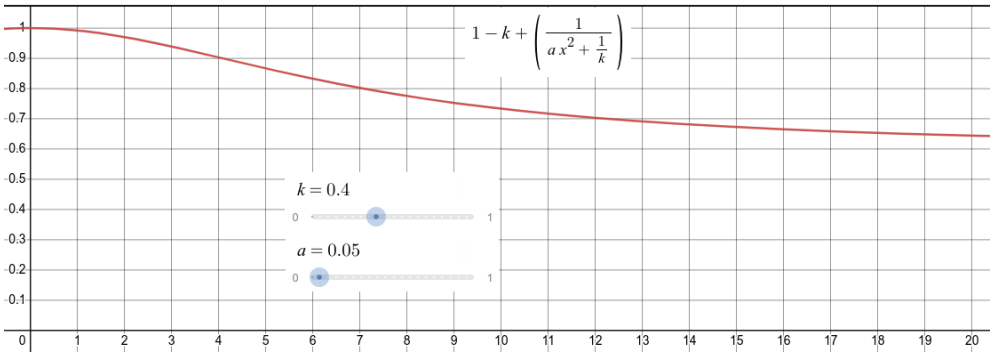

Figure 4.6: Age factor: The Y-axis is the multiplier applied to the score and X-axis is the age of the document in years. Within the first year, the age factor is close to 1 and then it gradually approaches the discount $k = 0.4$ i.e. a factor of 0.6 as x increases.

As discussed in Section 4.1, the recommendation query is comprised of multiple text sources, most commonly the task title and description. Candidates that match both the title and the description receive a higher rank by summing the score contributions. Let $S_Q = (Q_1, \ldots Q_n)$ be a sequence of different query sources vectorized into the representation that the selected ranking method requires. For each top candidate document D, based on the initial index side ranking, the scores of multiple query matches are aggregated as follows:

$$\text{Score}(D) = \sum_{i=1}^{n} \text{Vector Rank}(Q_i, D) \cdot \text{Age Factor}(D).$$

The "Vector Rank" is defined by the selected ranking method (VSM, BM25 or SBERT). Other aggregation strategies such as taking the maximum or average were experimented with, but they resulted in a lowered recall.

After applying the age factor and aggregating multiple query source hits, filters based on the document type are applied. For example, when the task is to find a solution to an error message, the commit descriptions can be useful

to find a module repository that should be updated. However, displaying multiple commit links to the same module repository is often redundant. Therefore, only the best scoring commit message per module is displayed as a result and other commit messages on the same repository are ignored. This prohibits multiple commits with similar descriptions such as "fix ansible connector" to not dominate the entire result list when doing a query such as "ansible connector". The configuration of redundancy filters based on source field type is done on the Gather data model that defines the index sources (discussed in Section 3.1).

Finally, as an optional stage, the BERT cross-encoder can be used to further re-rank the top-k results that will be returned to the user. This stage does not affect the *recall@k* instead, it only reorders the results displayed to the user. The effectiveness of reranking is measured with a metric called *nDCG@k* introduced in Section 2.2. The next chapter discusses the most important performance optimizations related to this ranking pipeline.

4.4 Performace Optimizations

The focus of this section is on the performance optimizations that affect BM25 as it was the chosen method for production use based on the effectiveness evaluation discussed later in Chapter 5. This section starts by briefly discussing the performance of preprocessing and why it is important. The rest of this section focuses on how the inverted index, stored in the PostgreSQL database, is used effectively. Two example SQL queries are used to illustrate encountered performance bottlenecks: one example that is naive and inefficient and an improved example that achieves a 55-fold speedup compared to the naive approach.

Optimizing performance starts by precomputing as much as possible using background processes. In addition to precomputing the document representations during indexing, statistics used in the retrieval such as the token IDF, euclidean norm and the average document length, are periodically computed and stored. The computational performance of the indexing process is important for two reasons. First, during retrieval, the query is transformed and indexed with the same procedure as the existing documents in the index. Thus, the latency of indexing one text document is combined with the query response time. The second reason is related to development efficiency: If the preprocessing or tokenization is improved, the affected documents need to be reindexed for the improvements to come into effect. For these reasons, the indexing uses a minimal amount of SQL queries, as writing to the database remains the bottleneck for indexing performance.

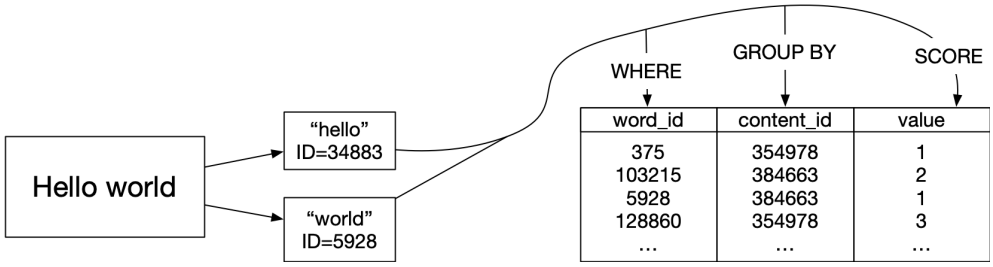

Figure 4.7: The inverted index as an associative database table. The word tokens of the query text are used as the WHERE-domain in the retrieval SQL query. The results are grouped by the IR Content identifier (*content_id*) and scored based on the associated value. The value is the word frequency or the depth for stack trace tokens. Otherwise, the retrieval process is identical for word and stack trace tokens.

For sparse vector methods, using an inverted index is the basis for performance, as it quickly prunes the set of possible candidates. The inverted index mapping from word tokens (*word_id*) to documents (*content_id*) is illustrated in Figure 4.7. In addition to functioning as an inverted index, the same database table stores the term vector representation used by VSM and BM25.

Even though BM25 and VSM are conceptualized as inner products of term vectors, it does not make sense to store the full vectors as arrays. Consider SprintIT's searchable text index that consists of 320000 unique words and, on average, 26 non-zero word frequencies per document. With these statistics, the average array vector representing a document would have to store $320000 - 26$ zeros that do not even contribute to the score. For this reason, only the association between a document (*content_id*) and a word token (*word_id*) is stored, along with an additional *value* column that represents the token weight. The additional column *value* is used for ranking which stores the token frequency for words and the depth for stack trace tokens (see Section 3.4). Figure 4.7 illustrates how this associative database table is used during retrieval.

After aggregating the *value* column and other pre-computed statistics into a score (as defined by the selected method VSM or BM25), the results are sorted based on the score. A limited set of the highest-scoring results are retrieved from the index to the application side. The limit is an important performance factor as converting large amounts of data from PostgreSQL to Python is costly and unnecessary since only the top results are needed. Based on experiments, a limit of 5000 top candidates is more than sufficient

to not have the approximation of the score affect the recall.

```
# Python side one-liner that uses the inverted index to map approximately
# 200-word IDs to 100 000-candidate IDs through the methods of Odoo ORM.
candidate_ids = query_texts.mapped('word_ids.content_ids').ids
```

```
    SELECT
        c.data_field_id AS data_field_id,
        c.owner_model AS owner_model,
        c.id AS content_id,
        c.doc_len AS doc_len,
        c.data_create_date AS data_create_date,
        t.id AS token_id,
        t.idf AS idf,
        COALESCE(g.weight, 1.0) * q.value AS value
    FROM ir_recommend_content_ir_recommend_word_rel q
        INNER JOIN ir_recommend_content c ON c.id = q.ir_recommend_content_id
        LEFT JOIN ir_recommend_field_weights g ON (
            g.data_field_id=c.data_field_id AND parameter_id=1
        )
        INNER JOIN ir_recommend_word w ON q.ir_recommend_word_id=w.id
        INNER JOIN ir_recommend_stemword t ON w.stemmed_word_id=t.id
    WHERE
        c.id IN (
            candidate_ids = <approximately 100 000 content IDs>
        )
        AND w.active=True
```

Figure 4.8: A naive SQL query with multiple performance bottlenecks. The "WHERE" domain, highlighted in the figure, is directly on the document table "c" because the mapping from approximately 200-word tokens into approximately 100 000 document IDs is done beforehand, outside of the SQL query. Scanning the document table with a list of 100 000 IDs is considerably slower than scanning the word token table with only 200 IDs, even though the resulting set of candidates is identical. In addition to being slow, this query has the possibility of hitting the memory limit of the calling Python process, as it will always return the entire subset of documents, matched by the inverted index. A ranking score is not approximated on the index side, thus there is no way of further limiting the results that are loaded to the application side. The average time for generating recommendations with this approach is 110 seconds.

To illustrate the importance of using the inverted index correctly, let's consider two example SQL queries using the ranking method BM25. Figure 4.8 displays a naive SQL query that fetches the document metadata and the *value* column for calculating the score on the application side. In this approach, the inverted index mapping from query word tokens to a subset of candidate documents is done before the SQL query, with a one-liner method

```
# Python side: select a proper domain for the inverted index, based on word tokens.
word_ids = query_texts.mapped('word_ids').ids
```

```
     SELECT
        c.data_field_id AS data_field_id,
        c.owner_model AS owner_model,
        c.owner_id AS owner_id,
        c.id AS content_id,
        c.doc_len AS doc_len,
        c.data_create_date AS data_create_date,
        ARRAY_AGG(t.id) AS token_ids,
        ARRAY_AGG(COALESCE(g.weight, 1.0) * q.value) AS values,
1.      SUM( t.idf *  (
            q.value /
            (q.value + 1.2 * ( 1 - 0.75 + ( 0.75 * (c.doc_len / 62.49)))))
        ) AS candidate_order
    FROM ir_recommend_content_ir_recommend_word_rel q
        INNER JOIN ir_recommend_content c ON c.id = q.ir_recommend_content_id
        LEFT JOIN ir_recommend_field_weights g ON (
            g.data_field_id=c.data_field_id AND parameter_id=1
        )
        INNER JOIN ir_recommend_word w ON w.id = q.ir_recommend_word_id
        INNER JOIN ir_recommend_stemword t ON t.id = w.stemword_id
    WHERE
2.      t.id IN (
            word_ids = <approximately 200 word token IDs>
        )
        AND w.active=True
    GROUP BY c.id
3.  ORDER BY candidate_order DESC
    LIMIT 5000
```

Figure 4.9: Improved SQL query that eliminates the performance bottlenecks described in Figure 4.8. The first highlighted section of the query computes an approximation of the final BM25 score. The approximate score is used in the third highlighted section to sort the results, which allows limiting to 5000 candidates that are further processed on the Python side. The second highlighted section shows a better domain compared to the highlighted section of Figure 4.8 that directly limits the resulting candidates with the array of word token IDs instead of converting to document IDs beforehand. The average time for generating recommendations with this approach is 2 seconds which is a 55-fold speedup compared to the approach described in Figure 4.8.

of Odoo ORM. The resulting 100 000 candidate IDs are used as a WHERE domain for fetching the data needed for ranking. Using the Odoo ORM to first map 200 query tokens into 100 000 document IDs and then doing another query on the same table using the 100 000 document IDs makes no sense from a performance perspective. Still, the mistake is easily done with the high-level methods provided by the Odoo ORM.

For comparison, Figure 4.9 illustrates a better example SQL query that properly utilizes the inverted index. Instead of leaving the whole ranking process to the application side, the SQL query in Figure 4.9 calculates an approximation of the final score for the candidate documents and fetches only the top 5000 candidates. In addition, the improved query applies the 200 query tokens directly as a domain as opposed to first mapping them to document IDs. Better usage of the inverted index results in significant performance gains: the average elapsed real-time for generating recommendations decreased from 110 seconds to 2 seconds, which is a 55-fold speedup. Note that this speedup was measured on the production server that uses shared cloud platform, while the evaluation results that will be presented in Chapter 5 were measured on a local workstation.

The discussed performance bottlenecks were discovered using a test bench that is tightly integrated into the system. Section 4.5 discusses the test bench included as part of the production system and describes its functionality.

4.5 Test Bench

Robustly developing the retrieval system is difficult as the information needs of the user have to be satisfied on a large set of varying data and queries. Often an improvement on one input query will result in decreased performance for other queries. For this reason, systematically evaluating improvements on annotated test data is essential to facilitate robust changes. This section describes the functionality of the test bench, included in the implementation and how it can be used in future development.

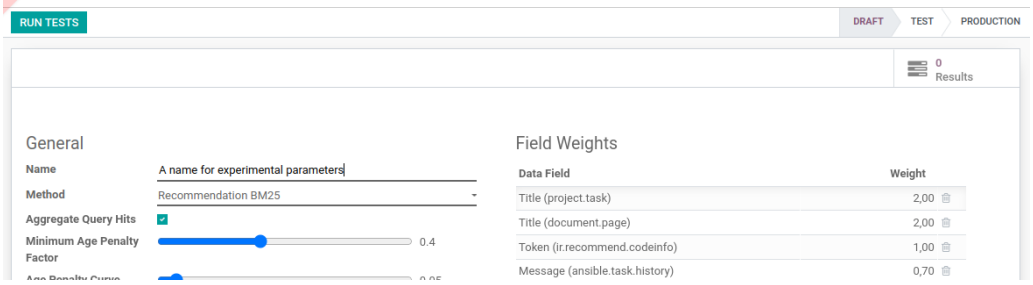

Figure 4.10: A view for defining experimental parameters and running the test cases. After selecting the desired retrieval method and related parameters, the button “Run Tests” in the top-left corner can be used to evaluate the selections by running the retrieval pipeline on all the collected test cases. A link to the test results is provided to the user on the top-right corner.

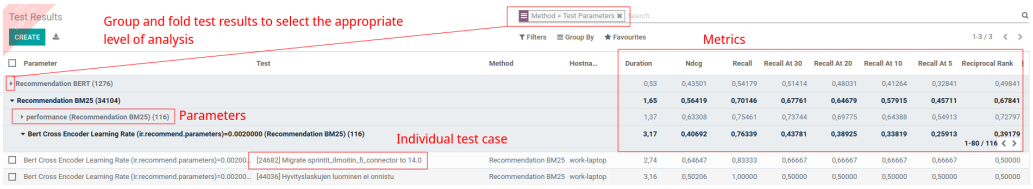

Figure 4.11: A flexible view to analyze test results. Grouping on the selected fields aggregates the average of all the metrics displayed on the right. The view allows drilling down to individual test result levels. Clicking on the individual test result allows for browsing through the retrieved documents and inspecting directly what is behind the numbers.

When a user annotates a recommendation or search result with a relevance judgement, a test case for that information need is automatically generated. In addition, the system also allows manually defining test cases without a relevance judgement when an issue is identified on the system. After a set of test cases are identified, the retrieval pipeline with a selected retrieval method and parameters can be evaluated with respect to the test cases. Figure 4.10 shows the view for selecting parameters and executing test cases.

After executing the tests on the selected method and parameters, the effectiveness measures, defined in Section 2.2, are computed and the results are stored for future reference. Figure 4.11 shows the aggregate view for analyzing test results. Standard Odoo allows defining flexible views where the test results can be grouped and filtered to make the appropriate analysis and comparisons. To quickly identify the test cases that produce a deviation with two different parameter configurations, an additional "Compare" feature is implemented that automatically filters out the test cases that had identical results with both sets of parameters. Keeping track of the previous test results with different configurations and implementation versions aids in detecting and locating issues introduced by new development work. Using the test bench features allows for robust development of the system in the future.

Figure 4.12 shows the view for defining parameter experiments. Parameter experiments are used to automatically generate a specified range of parameter values and run the test cases for the whole range. If the system administrator wishes to change a parameter, an experiment on the effects of the change can be done with the following steps:

1. Define a parameter experiment by selecting a parameter field and a parameter sweep range as displayed in Figure 4.12. The range should

include the parameter value currently in production use such that you can view how changes from the status quo would affect performance.

2. Generate the different parameter configurations.
3. Run the tests for the parameter range generated and plot the results.
4. Select two or more parameter definitions and use the "Compare" action to inspect further what recommendations and search results are behind the measured changes.
5. Use the gained information to evaluate whether the parameter should be changed.

Using this workflow, the system effectiveness and appropriate parameter selections were evaluated. Chapter 5 details the evaluation experiments and overviews the results.

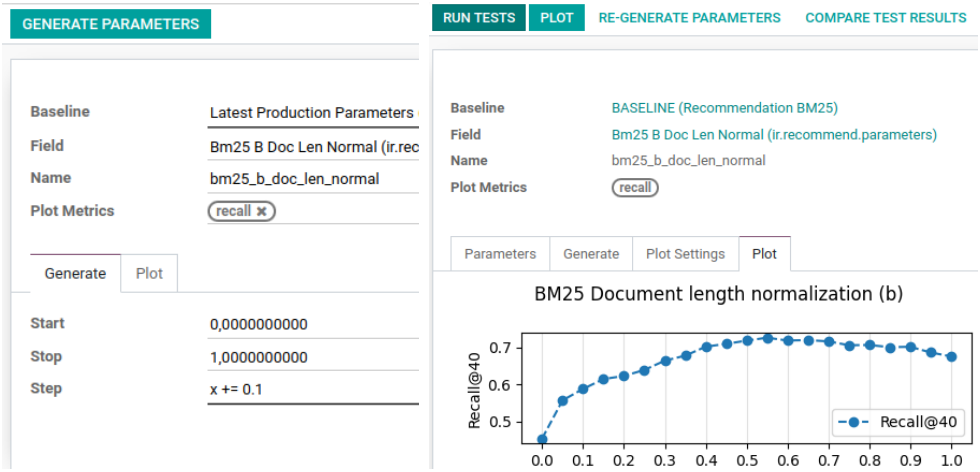

Figure 4.12: The user interface for defining parameter experiments. First fill in the desired parameter on "Field" and then define a range by filling in "Start", "Stop" and "Step". Then press "Generate Parameters" and "Run Tests". After the tests finish executing, the results for selected metrics can be plotted with "Plot" button.

Chapter 5

Evaluation

The test bench described in Section 4.5 is used to perform a quantitative evaluation of the retrieval effectiveness and efficiency. Regarding efficiency, this chapter reports wall-clock latencies from the beginning of the retrieval process to the resulting documents presented to the user. The effectiveness evaluation is done based on manually assigned relevance judgements. The dataset used for evaluation consists of 116 test cases that are a mix of tasks and ad hoc queries. The retrieval results for these test cases are manually annotated resulting in a total of 662 relevant links and 894 text pairs marked as relevant. There are more relevant text pairs than links since multiple different texts can originate from multiple fields of one document in Odoo. Metrics defined in Section 2.2 are considered in the evaluation.

Section 5.1 enumerates the values used for the free hyperparameters and the experiments made to explore the parameter space. Section 5.2 reviews the results of the experiments and compares the results of different methods. Section 5.4 summarizes discussions with a tester group and other users of the system. Finally, Section 5.5 discusses the robustness of the system.

5.1 Hyperparameters and Experiments

Comprehensive parameter optimization is left out of the scope of this thesis due to a time constraint. Instead, default hyperparameter values are selected based on previous research and then adjusted within reasonable limits based on experiments. The experiments are made by doing one-dimensional parameter sweeps while keeping the other parameters at the default value. Future possibilities for more advanced parameter optimization are discussed in Section 6.2.

As discussed in Section 2.4, it is common to use values of $b = 0.75$ and

$k_1 = 1.2$ for the BM25 formula in the absence of optimization. Sun et al. [59] arrived at the value of $k_3 \approx 0.4$ after optimization thus it is selected as an initial value in this thesis. The field weights as well as the weights for different stack trace features have a default value of 1. The VSM has no inherent parametrization except for selecting the normalization of term frequencies. This thesis uses the standard logarithmically scaled frequency proposed in the initial VSM paper by Salton et al. [56] and leaves experiments with different normalization schemes for future research. Since the BM25 model produced the most promising results with default parametrization, it has the most experiments defined concerning parameter fine-tuning. See Table 5.1 for a complete list of used BM25 parameters.

Table 5.1: Parameter experiment ranges for BM25 defined by the Start, Stop and Step columns. When experimenting on one of the parameters, the default value is used for all the other parameters. The final column has the parameter value selected for production use. Compared to the default parametrization, the final production parametrization increased the overall average recall from 0.706 to 0.763.

Parameter Name	**Default**	**Start**	**Stop**	**Step**	**Production**
Document length normalization (b)	0.75	0.00	1.00	0.05	0.55
Candidate term saturation (k_1)	1.20	0.00	3.00	0.10	1.20
Query term saturation (k_3)	0.40	0.00	3.00	0.10	0.60
Age penalty factor (k)	0.60	0.00	1.00	0.10	0.40
Age penalty slope (a)	0.15	0.00	1.00	0.05	0.05
Task title weight	1.00	0.00	2.00	0.10	1.30
Document Page weight	1.00	0.00	2.00	0.10	2.00
Stack Trace Token weight	1.00	0.00	2.00	0.10	1.00
Ansible Error weight	1.00	0.00	2.00	0.10	0.70
Attachment weight	1.00	0.00	2.00	0.10	0.80
Stack Trace Error Phrase Feature	1.00	0.00	20.00	2.00	8.00
Stack Trace Line of Code Feature	1.00	0.00	20.00	2.00	1.00
Stack Trace Function Name Feature	1.00	0.00	20.00	2.00	0.00

For the SBERT bi-encoder, the initial model checkpoint selected is called *multi-qa-MiniLM-L6-cos-v1*[1] available on the *Hugging Face* model hub [67]. It is based on a 6-layer version of the distilled MiniLM model [65] and fine-tuned on 215 million pairs of questions and answers from diverse sources. The model produces embeddings of size 384 from variable length input sequences. The model checkpoint was selected since it was designed for semantic search and trained on pairs of questions and answers which closely resembles our problem of finding solutions to customer problems. The model also has a high max sequence limit (512) that facilitates the highly varying document size in SprintIT search index. In addition, the model was one of the most popular models in *Hugging Face* model hub [67] based on monthly downloads.

[1]https://huggingface.co/sentence-transformers/multi-qa-MiniLM-L6-cos-v1

The bi-encoder model is further fine-tuned with *Multiple Negatives* loss using mean-pooling and cosine similarity as a ranking function with a scale of 20. The fine-tuning procedure and default hyperparameters are identical to how the checkpoint was trained, except for batch size and warm-up steps. A typical batch size for BERT-based models is from 32 to 64, but due to GPU memory constraints, a batch size of 8 is used in this thesis. This decreases the accuracy of the bi-encoder as the Multiple Negatives loss benefits from a larger batch size (see Section 2.5). The original model checkpoint was trained with 1000 linear warm-up steps, but since the training set of this case study is smaller, a warm-up of 10% of the training data is used, as recommended in [43]. The effect of changing the learning rate and the number of epochs from the default values is experimented with. A learning rate of $2 \cdot 10^5$ and ten epochs are used as the default value in parameter sweep experiments.

For the cross-encoder, a model checkpoint called *ms-marco-MiniLM-L-12-v2*[2] is used as the initial starting point. Unlike the bi-encoder, the model is based on the original 12-layer version of distilled MiniLM model [65] and fine-tuned on the *MS MARCO Passage Ranking* task [38]. Again, the original fine-tuning procedure is followed except for batch size and warm-up steps. Learning rate of $2 \cdot 10^{-5}$ and one epoch are used as a starting point for experiments. Both BERT-based models, i.e the cross-encoder and the bi-encoder have an inherent limit of 512-word pieces and longer sequences are truncated automatically.

For both the cross-encoder and the bi-encoder BERT models, a random holdout split of 20% (23/116) is used to assess generalization. For both BERT models, the first experiment is executed without fine-tuning on SprintIT data and relying only on the original fine-tuning of the model checkpoints. The second experiment is to fine-tune the models with SprintIT data and the default parameters based on the original model fine-tuning discussed earlier in this section. Then, similar parameter sweeps as with BM25 are conducted on learning rate and epochs. Learning rates $\{2 \cdot 10^{-6}, 2 \cdot 10^{-5}, 2 \cdot 10^{-4}, 2 \cdot 10^{-3}\}$ are experimented with both the bi- and the cross-encoder, and the epoch values are $\{5, 10, 20, 40, 80\}$ and $\{1, 2, 4, 8, 16\}$ for the bi- and the cross-encoder respectively. The effect of the cross-encoder is evaluated only as a re-ranker for the top 40 results of the best-performing method (BM25). The aim is to evaluate if the BERT cross-encoder can bump the relevant documents higher in the results, saving the user browsing time.

All these experiments result in 292 different parameter configurations with $292 \cdot 116 = 33872$ distinct test executions. The combined retrieval time for all the test executions is 15 hours with an average retrieval time

[2]https://huggingface.co/cross-encoder/ms-marco-MiniLM-L-12-v2

of 1.6 seconds per test execution. Training the SBERT and indexing the SBERT embeddings to Faiss are not included in these calculations. All tests are executed sequentially on a home workstation with an 11th Gen Intel(R) Core(TM) i7-1165G7 @ 2.80GHz CPU and a NVIDIA GeForxe GTX 1060 6GB GPU. Experiments with VSM and BM25 use only the CPU through PostgreSQL, while BERT experiments utilize the GPU. Section 5.2 overviews the results of these experiments.

5.2 Experiment Results

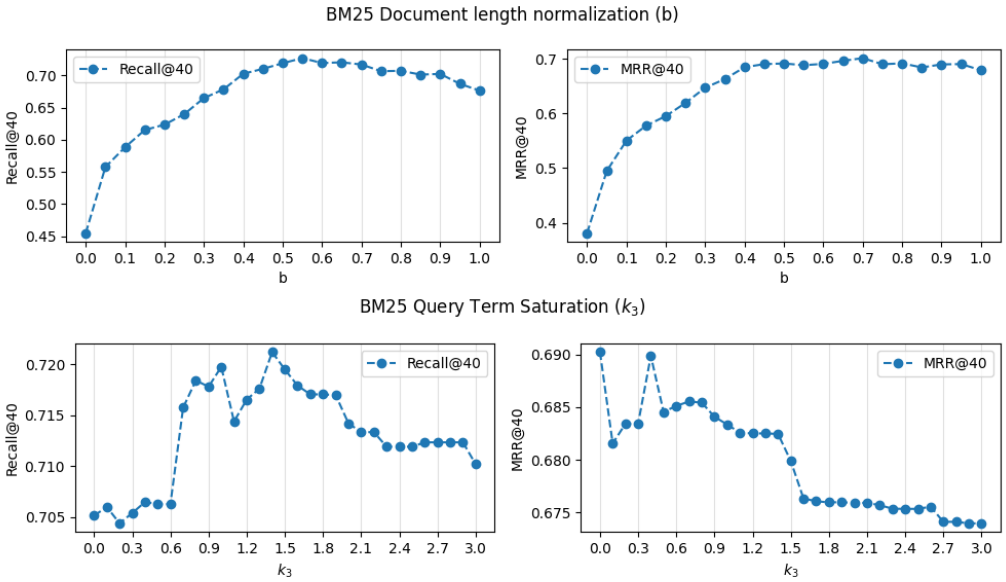

Figure 5.1: Example result graphs of BM25 parameter sweep experiments with b and k_3 as example parameters and average *recall@k* and *reciprocal rank@k* as metrics. The graphs on the right column measure *recall@k* (y-axis) and the left column shows the measurements of *mean reciprocal rank@k* (MRR) with $k = 40$. The first row of graphs shows the effect of varying the BM25 parameter b which controls the importance of document length normalization (x-axis). Based on both recall and MRR, a value of $b = 0.55$ is a reasonable choice for b. The second row of graphs shows the effect of varying the BM25 query term saturation parameter k_3 (x-axis). The recall plot on the bottom-left corner suggests that $k_3 = 1.4$ would produce the best result while the mean reciprocal rank plot on the bottom-right corner suggests lower values for k_3.

The system allows plotting the experiment results on graphs as shown in Figure 5.1. All the measurements are averages across the 116 test cases. Due to the small dataset size and the simple parameter sweep experiment setup, the resulting graphs should not be considered definitive, but rather a guide for selecting reasonable parameters for the production system. Other factors are also considered when selecting the production parameters, such as recommendations from existing literature and domain knowledge of the system. For example, it is known that encountered deployment errors in the Ansible system do not have direct solutions but only provide at best the name of the employee who encountered the error. Thus, they are weighted lower than other document sources.

The first row of graphs in Figure 5.1 shows the effect of BM25 parameter b that controls document normalization. As discussed in Section 2.4 there exists a significant amount of previous research that suggest a value of $0.5 \leq b \leq 0.8$ works well in most circumstances. Based on this knowledge combined with the graphs in Figure 5.1 a value of $b = 0.55$ is selected for production use. The second row of graphs in Figure 5.1 shows the effect of the BM25 parameter k_3 that controls query term saturation. A value for the BM25 parameter $k_3 = 0.6$ is selected after considering the results displayed in Figure 5.1 and the optimization result of [59] $k_3 \approx 0.4$. Selecting the BM25 parameter k_3 confidently is difficult because the parameter lacks the volume of experiments from previous research compared to the standard parameters b and k_1 of BM25. In addition, there are no external justifications for selecting k_3, contrary to the example of selecting the field weight of Ansible deployment errors discussed earlier in this section.

Table 5.2: Average recall rates at k for different retrieval methods with varying k. The best results are bolded. For SBERT the test case results are split according to the 80/20 train-test split and for other methods, the average recall values are computed across all the 116 test cases. BM25 has the best recall out of the three retrieval methods.

Top list size (k)	**VSM (tf-idf)**	**BM25**	**SBERT (train 80%)**	**SBERT (test 20%)**
5	0.176	**0.555**	0.362	0.486
10	0.223	**0.646**	0.438	0.446
20	0.286	**0.702**	0.536	0.397
30	0.327	**0.747**	0.583	0.322
40	0.373	**0.763**	0.615	0.510

After running the tests on the selected production parameters, the av-

erage recall rates across different methods are compared in Table 5.2. The recommendation list in production use is of length $k = 40$, but the recall values for $k \in \{5, 10, 20\}$ are also presented in Table 5.2. Since the amount of relevant candidates for each test case is quite small, the recall metric is the best indicator of effectiveness. Table 5.2 as well as the manual inspection of test case results, suggests that BM25 is the best choice for the retrieval method used in production.

Table 5.3: The average, minimum and maximum retrieval durations in seconds with different retrieval methods. SBERT has the best performance as it can utilize the GPU while the other retrieval methods run on the CPU.

	VSM (tf-idf)	**BM25**	**SBERT**	**BM25 + Cross-encoder**
avg	2.28 s	1.65 s	**0.53 s**	2.64 s
min	0.43 s	0.44 s	**0.20 s**	1.34 s
max	8.78 s	5.76 s	**0.71 s**	6.81 s

A comparison between the response times across all methods is presented in Table 5.3. SBERT has the best performance as it efficiently utilizes the GPU through Faiss. The variations in the minimum and maximum response times are directly linked to the query size: short ad hoc queries are fast while generating recommendations based on long task descriptions is slower. Longer retrieval time is acceptable for recommendations as they are generated with background processes, compared to ad hoc user queries that require low latency. Re-ranking the top 40 results of BM25 with BERT cross-encoder on GPU increased retrieval time by approximately 1 second on average regardless of the query size. When the test case task has a long description, sparse vector methods VSM/BM25 scale worse than SBERT. If long queries in the chosen production method (BM25) become an issue in the future, the recommendation query can be constrained to the top-20 highest IDF words as was done by Hiew [20] in his master's thesis. The difference between the VSM response time and BM25 can be explained by the prioritized optimization: VSM had an inferior recall compared to BM25, thus less time was spent on optimizing its runtime.

An overview of the experiment results from using the BERT cross-encoder to re-order the top 40 recommendations is displayed in Table 5.4. Although the cross-encoder was able to increase its performance after fine-tuning with the SprintIT data, the experiments suggest that using only BM25 has the highest recall in the test data set, implying the best generalization. A reasonable hypothesis, discussed further in Section 6.2, is that having more than

Table 5.4: The effect of using the cross-encoder as a re-ranker. The average *nDCG@40* values are reported separately for the train and test splits to assess generalization. The experiments with increased and decreased learning rates from $2 \cdot 10^5$ produced inferior results thus they are discarded from this table. In general, using only BM25 without the re-ranker produced the best results, even though the model trained with 16 epochs outperformed in the training dataset.

Model	**Train**	**Test**
BM25	0.644	**0.632**
BM25 + Cross-encoder no fine-tuning	0.460	0.453
BM25 + Cross-encoder fine-tuning with 1 epoch	0.550	0.561
BM25 + Cross-encoder fine-tuning with 16 epochs	**0.778**	0.612

116 test cases annotated could improve the performance of BERT models over the BM25 baseline.

5.3 The Effect of Stack Trace Tokens

This section discusses the effects of using the stack trace tokens as part of the document representation when computing BM25 score. All the graphs in this section have the feature weight on the x-axis. As discussed in Section 3.4, storing the frequency does not make sense for stack trace tokens as they rarely have repeating features. Instead, the depth of the token in the stack trace was stored during indexing and additional weight is assigned for each type of stack trace feature. The weight is treated similarly to the word frequency during ranking, as a stack trace token weight of x would correspond to a word frequency of x when computing the BM25 score.

Figure 5.2 illustrates how the stack trace error tokens improve both mean recall and mean reciprocal rank (MRR). Error phrases occur only at the bottom of the stack trace and are usually very specific to the given problem which explains the increased recall and MRR in Figure 5.2. Based on these results a value of 8 was selected for the error phrase feature weight.

Using the line of code features from stack traces increases recall, but lowers MRR as seen in Figure 5.3. Lines of code appear throughout the stack trace and the lines of code at the bottom of the stack are often identical for many different errors. Since large portions of the call stack are often shared across many unrelated errors, a high weight on the line of code feature decreases MRR. However, in some cases, the error phrase is identical for two different problems and the difference between two stack traces comes from

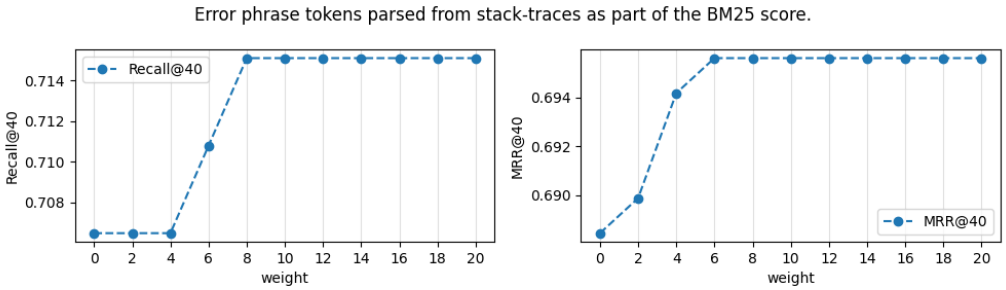

Figure 5.2: Recall and MRR gain from using error phrase tokens extracted from stack traces. The weight hyperparameter on the x-axis can be conceptualized as a hard-coded token frequency, when compared to the score contributions of word tokens.

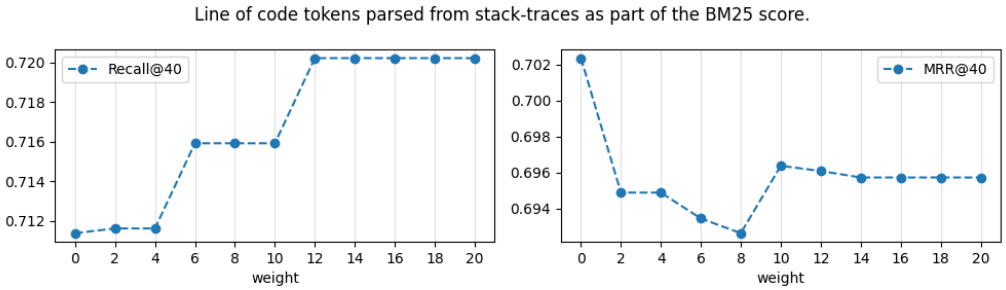

Figure 5.3: The compromise between recall and MRR when using the line of code feature extracted from stack trace.

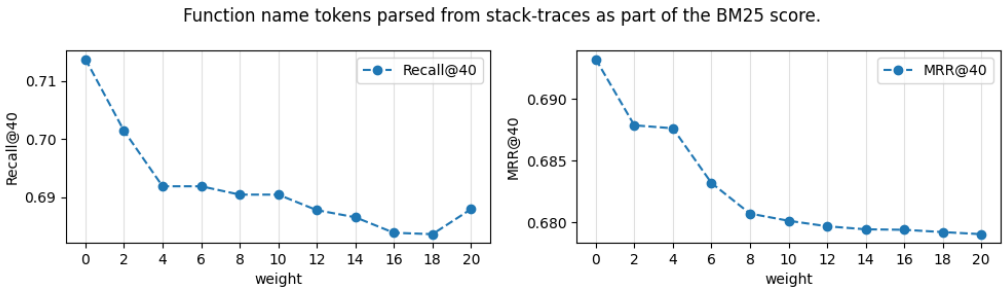

Figure 5.4: Decrease in recall and MRR when using function name features extracted from stack trace.

the lines of code that lead to the error phrase. To better distinguish between related and unrelated errors in these cases, a weight of one was selected for

the line of code stack trace feature.

Figure 5.4 shows that using the function name tokens parsed from stack trace degrades both recall and MRR. This suggests that using function names is not a specific enough feature for distinguishing between relevant and non-relevant stack traces. This is contrary to the results of Want et al. [66] which showed that using the canonical method signatures as tokens when detecting duplicate bug reports increases recall. Wang et al. [66] used a different method of recording the execution information and the programming language was Java instead of Python which has a different format for displaying the stack trace.

In addition to quantitative evaluation, discussions with a small test user group were held regularly to increase the probability that this thesis results in a useful product. Section 5.4 discusses the findings and comments made by the users.

5.4 Discussons with Users

Soon after an initial working prototype of the system was functional, a tester group was recruited to point out flaws, give comments and suggest improvements. The tester group had four users, but key takeaways from discussions with additional interested users are also included in this section.

The feedback from all the users was positive. The users agreed that the company needs a system for finding and discovering relevant information, especially as the organization grows. The users gave multiple examples where the recommendations had a direct link to a previous solution when faced with a problem from a customer. In addition, the ad hoc search was deemed particularly useful in cases where the user knew a solution exists but did not remember where to find it. One user compared the search functionality to the search offered by modern email clients. The users agreed that the system would increase the efficiency of their work. The users estimated that the solution recommender finds a helpful link for 10% of their tasks that reduces the solving time by 50%. If the user estimates generalize to the full scale of the company, the system could potentially save a significant amount of unnecessary work. As some of the work is invoiceable, the benefit would also be shared with the customers.

Regarding the ad-hoc search, one test user wished that the system would give an empty list of results if no solution exists in the system as this would save browsing time. A possible future implementation of a threshold based on the score is discussed in Section 6.2. One user noted that for new module specifications, even not finding a previous implementation is useful, as it gives

a better justification for starting new development from scratch. Using the solution retrieval engine supplements the results from public search engines when trying to find previous solutions that match a customer specification.

The most common problem identified by the users forgetting to utilize the new tool in their daily work. Adopting new features often requires time before using them becomes an integrated part of the user's workflow. To ease the adoption, users suggested pop-up remainders or other ways of making the recommendations tab stand out from the task interface. We decided to address this by moving the ad hoc search box from the recommendations tab to the task header, where it attracts more attention and guides the user further investigate the recommendations tab.

The users identified some tasks where a solution could not be found with the implemented IR system. With some of the tasks, it was unknown if a previous solution even exists, whereas with other tasks the user knew that the same problem was solved previously but the system failed to retrieve it. The former cases were instances of the *vocabulary mismatch problem* where the words used to describe the issue were different on the solution compared to the query.

During the discussions, we identified a problem in getting relevance feedback from users. In one case, a user had a relevant recommendation that provided an answer to a customer question, but the user did not click the "thumbs up" button to annotate the result. Furthermore, click-through was not needed since the summary already gave the solution to the problem. Therefore, the interaction was not recorded by the system at all, leaving valuable relevance information on the table. At the time of writing, 27 user-assigned relevance judgements and 101 click-throughs are recorded in the system, excluding the records made by the author. To what extent the users will annotate helpful retrieval results in the future remains to be seen.

5.5 Discussion of Robustness

The subjective nature of relevance, discussed in Section 2.2, ought to be considered when analyzing the effectiveness results of Section 5.2. Since this case study started from an unannotated dataset, much of the relevance judgements are made by the author, in retrospect, after the task was already solved. In most cases, the results can be objectively identified as relevant, for example when the same exact error was found on multiple tasks. However, some retrieval results are annotated to be relevant even if the result contains only background information that the author deems useful in solving the problem. Of course, such background information is not relevant to an

expert who already knows the background and just needs a specific fix for the problem at hand.

The test set annotation process has a bias toward high recall: a large portion of the relevant documents were discovered by going through the results of different retrieval methods during development. Although thoroughly browsing multiple pages of retrieval results is a good way to discover relevant links that the system can produce, this does not provide an exhaustive list of false negatives not recommended by the system. As false negatives are important for improving the system, a considerable amount of time was spent to find false negatives by inspecting user-made internal links and by asking for help from colleagues if they know of similar tasks.

The effect of a larger search index was not explored in this thesis. It is reasonable to assume that when a haystack grows, finding a needle becomes harder. The age discount factor, described in Section 4.3, should help in pruning the searchable index from out-of-date results. In addition, with a growing amount of data and usage, new scaling bottlenecks that are not yet addressed can arise, even though efficiency was a major consideration in this thesis.

As discussed in Section 5.1 the parametrization is likely not optimal and may require adjustments in the future. Section 4.5 outlined a procedure for adjusting parameters in a controlled way. The text processing stages, particularly cleaning the emails and parsing the Python stack trace, have heuristical nature motivated by the examples found in production. Section 6.1 discusses these in more detail and gives suggestions for future improvements.

Chapter 6

Future research and conclusions

Multiple promising ideas were left out of the scope for this thesis due to a time constraint. This chapter discusses ideas for the future development of the system and provides a concluding section. Section 6.1 focuses on the indexing process described in Chapter 3. Section 6.2 discusses improvements to the implemented retrieval methods described in Chapters 2 and 4 and suggests research on other promising methods. Finally Section 6.3 gives concluding remarks.

6.1 Improving the Indexing

Adding new text sources to the search index is recommended. For example, the commit descriptions of third-party Odoo modules commonly used by SprintIT customers can contain solutions to customer problems. Fetching the source files from Bitbucket[1] and Github[2] can be useful for linking stack traces to a potential developer. In addition, SprintIT uses Google Drive extensively for documentation, thus integrating it with Odoo can further increase the significance of the implemented IR system.

Parsing the email signatures and replies relies on hand-crafted patterns. Research papers such as [13, 30, 44] have more sophisticated approaches that could be utilized in the future. Similarly parsing the stack trace tokens relies on hand-crafted features and more importantly, properly recovering the line breaks that separate the different features from each other. In a case where the linebreaks are lost, the stack trace is effectively ignored for the search. Having a structured web form for support requests would help in getting

[1] https://bitbucket.org/
[2] https://github.com/

cleaner task descriptions and stack traces without having to worry about email replies and signatures.

In natural language tokenization, lemmatization could be a better approach to removing inflexions from words compared to the more simple stemming algorithms used in this thesis. According to Korenius et al. [29] using lemmatization, which involves the splitting of the compound words, is a better normalization method than stemming for Finnish text documents due to the highly inflectional and agglutinative nature of Finnish language.

In this thesis, the language gap between tasks written in English and Finnish was not addressed. Although the initial experiments made during this thesis did not show increased recall after applying translation to the query texts, a more sophisticated approach for utilizing translations could allow content written in English to be recommended for tasks written in Finnish. The language gap is also problematic when stemming the word tokens as the stemming rules differ depending on the language. To address this issue, character n-grams could be used instead of tokenized words, removing the need for word normalization. McNamee et al. [34] found that using character n-grams with $n = 4$ exceeded the accuracy of using unnormalized words in bilingual retrieval and Sureka et al. [61] used n-grams for detecting duplicate bug reports.

Betterburg et al. [4] studied the question of "What makes a good bug report?" in 2008. They noted that an asymmetry exists between what the developers usually need from a good bug report, and what the users typically provide. To get more detailed information on bug reports, a crash reporter addon could be implemented to the systems of SprintIT customers that would automatically collect useful metadata such as the current view, latest backend call and the associated stack trace when an error arises.

6.2 Improving the Retrieval

Due to the asymmetry in the indexed documents, the current retrieval implementation does not score the candidate document as a whole but instead computes the score based on the contents of one source field. Efforts to implement this caused an increase in irrelevant results that drowned out the actual answers, thus resulting in a lower recall. Conceptually, however, it would make sense to assign a higher score for links that have matching tokens in multiple fields. More research is needed to implement a ranking model that fully utilizes the whole document structure. To implement this properly in BM25F, combining the fields should be done on the token vector level, and not at the final score level, to preserve the desirable properties

of BM25 [46]. A new data model that allows configuring different document structures can be a good approach for implementing this in a way that increases effectiveness.

A mechanism to notify users about potentially irrelevant results could allow saving unnecessary browsing time. For example, if the score is very low or if only a small portion of the query terms are contained in any of the results then a notification could be displayed to the user. A more aggressive approach would be to define a score-based threshold and display only the results that exceed this threshold. However, it will probably be difficult to define a threshold in such a way that only irrelevant results are discarded.

To bridge the vocabulary gap discussed in Section 2.4, pseudo relevance feedback could be utilized as a query expansion mechanism. Pseudo relevance feedback assumes that the top-k retrieved results are relevant, adds them as part of the query and then re-executes the retrieval with the expanded query. According to Büttcher et al [10], pseudo relevance feedback can result in substantial improvements in effectiveness. However, there exists a risk that the top-k results contain words that drift the query away from relevant results. In addition to relevance feedback, the results could have a "More Like This" feature that would do a new search with the text of the previous result as the new query.

Another approach to alleviating the vocabulary mismatch problem is the query expansion by using a synonym database such as WordNet [36]. Lu et al. [33] developed a query expansion technique for a code search system based on WordNet that increased the precision and recall of the previous state-of-the-art technique by 5% and 8% respectively.

As discussed in Section 5.1, a proper hyperparameter optimization was left for future research. Sun et al. [59] optimized the parameters of BM25F for duplicate bug report detection by using stochastic gradient descent inspired by the work of Taylor et al. [62]. A larger set of test cases and 10-fold cross-validation is recommended for automated parameter optimization. Importing a known dataset for duplicate bug report detection could help in the validation of generality. For this purpose, annotated datasets from the Eclipse, NetBeans or Open Office bug repositories are commonly available [31].

To improve relevance, proximity ranking could be added where distinct terms appearing close together in the query, get a higher rank if they are also close together in the candidate document. The current "bag of tokens" representation discards the positional information of tokens completely, which means that this approach would require a different document representation. Closely related to proximity ranking is the concept of phrase search where the user gives multiple words usually encapsulated in quotation marks, to require matching the exact phrase in the resulting documents. The prox-

imity operator $< - >$[3] of PostgreSQL can serve as a useful starting point for research of both the proximity ranking method and the phrase search functionality. In addition to phrase search, additional syntax for common operators such as AND, OR and NOT could be implemented.

Topic modelling is a common approach in mining software repositories [11]. Especially LDA, discussed briefly in Section 2.6, is a commonly used ranking factor in DBRD. Topic modelling could also be used for clustering the results and providing the user with higher-level topics to choose from or to filter with.

As the amount of user-provided relevance judgements and click-through records grows, the opportunities for supervised learning methods increase. The approach of fine-tuning the pre-trained BERT can become superior after enough data for fine-tuning has been accumulated. As more sophisticated neural network-based models are introduced, the work done in this thesis is a solid starting place for utilizing the new models.

The success of graph models such as PageRank [40] in web page retrieval, makes experimenting with similar approaches an interesting path for future research. For example, the click-through records and relevance judgements could be interpreted as edges in a graph with some weights associated with them. Another approach could be to use the top results retrieved by BM25 as the edges in the graph with the BM25 score as the weight on the edges.

6.3 Conclusions

In this thesis, an efficient IR system was implemented and shown to aid the employees of SprintIT in finding previous solutions to their tasks. When a new task is submitted, the system provides recommendations to internal documents that contain relevant information and previous solutions. In addition, the system allows the employee to make ad hoc keyword queries to rapidly find information from a large collection of indexed documents.

Gathering all the relevant documents from different parts SprintIT Odoo into a single search index facilitated the implementation of a system-wide IR engine. A satisfactory retrieval latency was achieved by tackling performance bottlenecks of the index implementation. Although the system was developed for the internal use of SprintIT employees, the design is configurable and easily adaptable to be used in future customer projects as well.

Three retrieval models called VSM, BM25 and BERT were implemented to retrieve relevant documents from the index. To evaluate the effectiveness,

[3] https://www.postgresql.org/docs/current/textsearch-controls.html

a test bench and an annotation tool were implemented as part of the system. The annotation tool collects relevance judgements and click-through data, used to evaluate the system and to facilitate future development. Out of the three evaluated methods, BM25 produced the highest recall of 76%. The separate processing of Python stack traces was shown to increase the recall of duplicate bug reports. Although BM25 was the most effective method, the BERT-based retrieval method achieved surprisingly good results, considering the small amount of annotated training data that was used for fine-tuning.

In this thesis, an effective information retrieval system was implemented and multiple paths for future research were identified. As a result of this thesis, SprintIT gained an IR system that allows employees to find the information they need and to utilize existing solutions to recurring problems.

Bibliography

[1] Aggarwal, K., Rutgers, T., Timbers, F., Hindle, A., Greiner, R., and Stroulia, E. Detecting duplicate bug reports with software engineering domain knowledge. In *22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering, SANER 2015, Montreal, QC, Canada, March 2-6, 2015* (2015), Y. Guéhéneuc, B. Adams, and A. Serebrenik, Eds., IEEE Computer Society, pp. 211–220.

[2] Ahasanuzzaman, M., Asaduzzaman, M., Roy, C. K., and Schneider, K. A. Mining duplicate questions in stack overflow. In *Proceedings of the 13th International Conference on Mining Software Repositories, MSR 2016, Austin, TX, USA, May 14-22, 2016* (2016), M. Kim, R. Robbes, and C. Bird, Eds., ACM, pp. 402–412.

[3] Alipour, A., Hindle, A., and Stroulia, E. A contextual approach towards more accurate duplicate bug report detection. In *Proceedings of the 10th Working Conference on Mining Software Repositories, MSR '13, San Francisco, CA, USA, May 18-19, 2013* (2013), T. Zimmermann, M. D. Penta, and S. Kim, Eds., IEEE Computer Society, pp. 183–192.

[4] Bettenburg, N., Just, S., Schröter, A., Weiss, C., Premraj, R., and Zimmermann, T. What makes a good bug report? In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2008, Atlanta, Georgia, USA, November 9-14, 2008* (2008), M. J. Harrold and G. C. Murphy, Eds., ACM, pp. 308–318.

[5] Bialecki, A., Muir, R., and Ingersoll, G. Apache lucene 4. In *Proceedings of the SIGIR 2012 Workshop on Open Source Information Retrieval, OSIR@SIGIR 2012, Portland, Oregon, USA, 16th August 2012* (2012), A. Trotman, C. L. A. Clarke, I. Ounis, J. S. Culpepper,

M. Cartright, and S. Geva, Eds., University of Otago, Dunedin, New Zealand, pp. 17–24.

[6] Bird, S., Klein, E., and Loper, E. *Natural Language Processing with Python: Analyzing Text with the Natural Language Toolkit*. O'Reilly Media, Inc., 2009.

[7] Blei, D. M., Ng, A. Y., and Jordan, M. I. Latent dirichlet allocation. *J. Mach. Learn. Res. 3* (2003), 993–1022.

[8] Budhiraja, A., Dutta, K., Reddy, R., and Shrivastava, M. DWEN: deep word embedding network for duplicate bug report detection in software repositories. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018* (2018), M. Chaudron, I. Crnkovic, M. Chechik, and M. Harman, Eds., ACM, pp. 193–194.

[9] Budhiraja, A., Dutta, K., Shrivastava, M., and Reddy, R. Towards word embeddings for improved duplicate bug report retrieval in software repositories. In *Proceedings of the 2018 ACM SIGIR International Conference on Theory of Information Retrieval, ICTIR 2018, Tianjin, China, September 14-17, 2018* (2018), D. Song, T. Liu, L. Sun, P. Bruza, M. Melucci, F. Sebastiani, and G. H. Yang, Eds., ACM, pp. 167–170.

[10] Büttcher, S., Clarke, C. L. A., and Cormack, G. V. *Information Retrieval - Implementing and Evaluating Search Engines*. MIT Press, 2010.

[11] Chen, T., Thomas, S. W., and Hassan, A. E. A survey on the use of topic models when mining software repositories. *Empir. Softw. Eng. 21*, 5 (2016), 1843–1919.

[12] Danilák, M. langdetect (python library). Retrieved from `https://github.com/Mimino666/langdetect`, 2014.

[13] de Carvalho, V. R., and Cohen, W. W. Learning to extract signature and reply lines from email. In *CEAS 2004 - First Conference on Email and Anti-Spam, July 30-31, 2004, Mountain View, California, USA* (2004).

[14] Deshmukh, J., Annervaz, K. M., Podder, S., Sengupta, S., and Dubash, N. Towards accurate duplicate bug retrieval using deep

learning techniques. In *2017 IEEE International Conference on Software Maintenance and Evolution, ICSME 2017, Shanghai, China, September 17-22, 2017* (2017), IEEE Computer Society, pp. 115–124.

[15] Devlin, J., Chang, M., Lee, K., and Toutanova, K. BERT: pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)* (2019), J. Burstein, C. Doran, and T. Solorio, Eds., Association for Computational Linguistics, pp. 4171–4186.

[16] Diaz, G. Finnish stopword list. Retrieved from `https://github.com/stopwords-iso/stopwords-fi`, 2016.

[17] Gormley, C., and Tong, Z. *Elasticsearch: the definitive guide: a distributed real-time search and analytics engine*. O'Reilly Media, Inc., 2015.

[18] Guo, J., Fan, Y., Pang, L., Yang, L., Ai, Q., Zamani, H., Wu, C., Croft, W. B., and Cheng, X. A deep look into neural ranking models for information retrieval. *Inf. Process. Manag. 57*, 6 (2020), 102067.

[19] Henderson, M. L., Al-Rfou, R., Strope, B., Sung, Y., Lukács, L., Guo, R., Kumar, S., Miklos, B., and Kurzweil, R. Efficient natural language response suggestion for smart reply. *CoRR abs/1705.00652* (2017).

[20] Hiew, L. Assisted detection of duplicate bug reports. Master's thesis, University of British Columbia, 2006.

[21] Hindle, A., Alipour, A., and Stroulia, E. A contextual approach towards more accurate duplicate bug report detection and ranking. *Empir. Softw. Eng. 21*, 2 (2016), 368–410.

[22] Indyk, P., and Motwani, R. Approximate nearest neighbors: Towards removing the curse of dimensionality. In *Proceedings of the Thirtieth Annual ACM Symposium on the Theory of Computing, Dallas, Texas, USA, May 23-26, 1998* (1998), J. S. Vitter, Ed., ACM, pp. 604–613.

[23] Isotani, H., Washizaki, H., Fukazawa, Y., Nomoto, T., Ouji, S., and Saito, S. Duplicate bug report detection by using sentence embedding and fine-tuning. In *IEEE International Conference on Software Maintenance and Evolution, ICSME 2021, Luxembourg, September 27 - October 1, 2021* (2021), IEEE, pp. 535–544.

[24] Jalbert, N., and Weimer, W. Automated duplicate detection for bug tracking systems. In *The 38th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2008, June 24-27, 2008, Anchorage, Alaska, USA, Proceedings* (2008), IEEE Computer Society, pp. 52–61.

[25] Johnson, J., Douze, M., and Jégou, H. Billion-scale similarity search with gpus. *IEEE Trans. Big Data 7*, 3 (2021), 535–547.

[26] Jones, K. S. A statistical interpretation of term specificity and its application in retrieval. *Journal of Documentation 28*, 1 (1972), 11–21.

[27] Jones, K. S., Walker, S., and Robertson, S. E. A probabilistic model of information retrieval: development and comparative experiments - part 1. *Inf. Process. Manag. 36*, 6 (2000), 779–808.

[28] Karpukhin, V., Oguz, B., Min, S., Lewis, P. S. H., Wu, L., Edunov, S., Chen, D., and Yih, W. Dense passage retrieval for open-domain question answering. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing, EMNLP 2020, Online, November 16-20, 2020* (2020), B. Webber, T. Cohn, Y. He, and Y. Liu, Eds., Association for Computational Linguistics, pp. 6769–6781.

[29] Korenius, T., Laurikkala, J., Järvelin, K., and Juhola, M. Stemming and lemmatization in the clustering of finnish text documents. In *Proceedings of the 2004 ACM CIKM International Conference on Information and Knowledge Management, Washington, DC, USA, November 8-13, 2004* (2004), D. A. Grossman, L. Gravano, C. Zhai, O. Herzog, and D. A. Evans, Eds., ACM, pp. 625–633.

[30] Lampert, A., Dale, R., and Paris, C. Segmenting email message text into zones. In *Proceedings of the 2009 Conference on Empirical Methods in Natural Language Processing, EMNLP 2009, 6-7 August 2009, Singapore, A meeting of SIGDAT, a Special Interest Group of the ACL* (2009), ACL, pp. 919–928.

[31] Lazar, A., Ritchey, S., and Sharif, B. Generating duplicate bug datasets. In *11th Working Conference on Mining Software Repositories, MSR 2014, Proceedings, May 31 - June 1, 2014, Hyderabad, India* (2014), P. T. Devanbu, S. Kim, and M. Pinzger, Eds., ACM, pp. 392–395.

[32] Lin, J., Nogueira, R., and Yates, A. *Pretrained Transformers for Text Ranking: BERT and Beyond*. Synthesis Lectures on Human Language Technologies. Morgan & Claypool Publishers, 2021.

[33] Lu, M., Sun, X., Wang, S., Lo, D., and Duan, Y. Query expansion via wordnet for effective code search. In *22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering, SANER 2015, Montreal, QC, Canada, March 2-6, 2015* (2015), Y. Guéhéneuc, B. Adams, and A. Serebrenik, Eds., IEEE Computer Society, pp. 545–549.

[34] McNamee, P., and Mayfield, J. Character n-gram tokenization for european language text retrieval. *Inf. Retr. 7*, 1-2 (2004), 73–97.

[35] Mezzetti, D. txtai (python library). Retrieved from `https://github.com/neuml/txtai`, 2020.

[36] Miller, G. A. Wordnet: A lexical database for english. *Commun. ACM 38*, 11 (1995), 39–41.

[37] Nguyen, A. T., Nguyen, T. T., Nguyen, T. N., Lo, D., and Sun, C. Duplicate bug report detection with a combination of information retrieval and topic modeling. In *IEEE/ACM International Conference on Automated Software Engineering, ASE'12, Essen, Germany, September 3-7, 2012* (2012), M. Goedicke, T. Menzies, and M. Saeki, Eds., ACM, pp. 70–79.

[38] Nguyen, T., Rosenberg, M., Song, X., Gao, J., Tiwary, S., Majumder, R., and Deng, L. MS MARCO: A human generated machine reading comprehension dataset. *CoRR abs/1611.09268* (2016).

[39] Nogueira, R., and Cho, K. Passage re-ranking with BERT. *CoRR abs/1901.04085* (2019).

[40] Page, L., Brin, S., Motwani, R., and Winograd, T. The pagerank citation ranking: Bringing order to the web. Technical Report 1999-66, Stanford InfoLab, November 1999. Previous number = SIDL-WP-1999-0120.

[41] Porter, M. F. An algorithm for suffix stripping. *Program 14*, 3 (1980), 130–137.

[42] Porter, M. F. Snowball: A language for stemming algorithms. Retrieved from `https://snowballstem.org/texts/introduction.html`, 2001.

[43] Reimers, N., and Gurevych, I. 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 2019, Hong Kong, China, November 3-7, 2019* (2019), K. Inui, J. Jiang, V. Ng, and X. Wan, Eds., Association for Computational Linguistics, pp. 3980–3990.

[44] Repke, T., and Krestel, R. Bringing back structure to free text email conversations with recurrent neural networks. In *Advances in Information Retrieval - 40th European Conference on IR Research, ECIR 2018, Grenoble, France, March 26-29, 2018, Proceedings* (2018), G. Pasi, B. Piwowarski, L. Azzopardi, and A. Hanbury, Eds., vol. 10772 of *Lecture Notes in Computer Science*, Springer, pp. 114–126.

[45] Robertson, S. E. The probability ranking principle in ir. *Journal of documentation 33*, 4 (1977), 294–304.

[46] Robertson, S. E., and Jones, K. S. Relevance weighting of search terms. *J. Am. Soc. Inf. Sci. 27*, 3 (1976), 129–146.

[47] Robertson, S. E., and Walker, S. Some simple effective approximations to the 2-poisson model for probabilistic weighted retrieval. In *Proceedings of the 17th Annual International ACM-SIGIR Conference on Research and Development in Information Retrieval. Dublin, Ireland, 3-6 July 1994 (Special Issue of the SIGIR Forum)* (1994), W. B. Croft and C. J. van Rijsbergen, Eds., ACM/Springer, pp. 232–241.

[48] Robertson, S. E., Walker, S., Jones, S., Hancock-Beaulieu, M., and Gatford, M. Okapi at TREC-3. In *Proceedings of The Third Text REtrieval Conference, TREC 1994, Gaithersburg, Maryland, USA, November 2-4, 1994* (1994), D. K. Harman, Ed., vol. 500-225 of *NIST Special Publication*, National Institute of Standards and Technology (NIST), pp. 109–126.

[49] Robertson, S. E., and Zaragoza, H. The probabilistic relevance framework: BM25 and beyond. *Found. Trends Inf. Retr. 3*, 4 (2009), 333–389.

[50] Robertson, S. E., Zaragoza, H., and Taylor, M. J. Simple BM25 extension to multiple weighted fields. In *Proceedings of the 2004 ACM CIKM International Conference on Information and Knowledge Management, Washington, DC, USA, November 8-13, 2004* (2004), D. A. Grossman, L. Gravano, C. Zhai, O. Herzog, and D. A. Evans, Eds., ACM, pp. 42–49.

[51] Rocha, H., de Oliveira, G., Marques-Neto, H., and Valente, M. T. Nextbug: a bugzilla extension for recommending similar bugs. *J. Softw. Eng. Res. Dev. 3* (2015), 3.

[52] Rocha, H., Valente, M. T., Marques-Neto, H., and Murphy, G. C. An empirical study on recommendations of similar bugs. In *IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering, SANER 2016, Suita, Osaka, Japan, March 14-18, 2016 - Volume 1* (2016), IEEE Computer Society, pp. 46–56.

[53] Rocha, T. M., and da Costa Carvalho, A. L. Siameseqat: A semantic context-based duplicate bug report detection using replicated cluster information. *IEEE Access 9* (2021), 44610–44630.

[54] Rogers, A., Kovaleva, O., and Rumshisky, A. A primer in bertology: What we know about how BERT works. *Trans. Assoc. Comput. Linguistics 8* (2020), 842–866.

[55] Runeson, P., Alexandersson, M., and Nyholm, O. Detection of duplicate defect reports using natural language processing. In *29th International Conference on Software Engineering (ICSE 2007), Minneapolis, MN, USA, May 20-26, 2007* (2007), IEEE Computer Society, pp. 499–510.

[56] Salton, G., Wong, A., and Yang, C. A vector space model for automatic indexing. *Commun. ACM 18*, 11 (1975), 613–620.

[57] Shuyo, N. Language detection library for java. Retrieved from `https://github.com/shuyo/language-detection`, 2010.

[58] Smith, R. An overview of the tesseract OCR engine. In *9th International Conference on Document Analysis and Recognition (ICDAR 2007), 23-26 September, Curitiba, Paraná, Brazil* (2007), IEEE Computer Society, pp. 629–633.

[59] Sun, C., Lo, D., Khoo, S., and Jiang, J. Towards more accurate retrieval of duplicate bug reports. In *26th IEEE/ACM International*

Conference on Automated Software Engineering (ASE 2011), Lawrence, KS, USA, November 6-10, 2011 (2011), P. Alexander, C. S. Pasareanu, and J. G. Hosking, Eds., IEEE Computer Society, pp. 253–262.

[60] Sun, C., Lo, D., Wang, X., Jiang, J., and Khoo, S. A discriminative model approach for accurate duplicate bug report retrieval. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE 2010, Cape Town, South Africa, 1-8 May 2010* (2010), J. Kramer, J. Bishop, P. T. Devanbu, and S. Uchitel, Eds., ACM, pp. 45–54.

[61] Sureka, A., and Jalote, P. Detecting duplicate bug report using character n-gram-based features. In *17th Asia Pacific Software Engineering Conference, APSEC 2010, Sydney, Australia, November 30 - December 3, 2010* (2010), J. Han and T. D. Thu, Eds., IEEE Computer Society, pp. 366–374.

[62] Taylor, M. J., Zaragoza, H., Craswell, N., Robertson, S., and Burges, C. Optimisation methods for ranking functions with multiple parameters. In *Proceedings of the 2006 ACM CIKM International Conference on Information and Knowledge Management, Arlington, Virginia, USA, November 6-11, 2006* (2006), P. S. Yu, V. J. Tsotras, E. A. Fox, and B. Liu, Eds., ACM, pp. 585–593.

[63] Tian, Y., Sun, C., and Lo, D. Improved duplicate bug report identification. In *16th European Conference on Software Maintenance and Reengineering, CSMR 2012, Szeged, Hungary, March 27-30, 2012* (2012), T. Mens, A. Cleve, and R. Ferenc, Eds., IEEE Computer Society, pp. 385–390.

[64] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I. Attention is all you need. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA* (2017), I. Guyon, U. von Luxburg, S. Bengio, H. M. Wallach, R. Fergus, S. V. N. Vishwanathan, and R. Garnett, Eds., pp. 5998–6008.

[65] Wang, W., Wei, F., Dong, L., Bao, H., Yang, N., and Zhou, M. Minilm: Deep self-attention distillation for task-agnostic compression of pre-trained transformers. In *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Process-*

ing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual (2020), H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, and H. Lin, Eds.

[66] Wang, X., Zhang, L., Xie, T., Anvik, J., and Sun, J. An approach to detecting duplicate bug reports using natural language and execution information. In *30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 10-18, 2008* (2008), W. Schäfer, M. B. Dwyer, and V. Gruhn, Eds., ACM, pp. 461–470.

[67] Wolf, T., Debut, L., Sanh, V., Chaumond, J., Delangue, C., Moi, A., Cistac, P., Rault, T., Louf, R., Funtowicz, M., Davison, J., Shleifer, S., von Platen, P., Ma, C., Jernite, Y., Plu, J., Xu, C., Scao, T. L., Gugger, S., Drame, M., Lhoest, Q., and Rush, A. M. Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations, EMNLP 2020 - Demos, Online, November 16-20, 2020* (2020), Q. Liu and D. Schlangen, Eds., Association for Computational Linguistics, pp. 38–45.

[68] Wu, Y., Schuster, M., Chen, Z., Le, Q. V., Norouzi, M., Macherey, W., Krikun, M., Cao, Y., Gao, Q., Macherey, K., Klingner, J., Shah, A., Johnson, M., Liu, X., Kaiser, L., Gouws, S., Kato, Y., Kudo, T., Kazawa, H., Stevens, K., Kurian, G., Patil, N., Wang, W., Young, C., Smith, J., Riesa, J., Rudnick, A., Vinyals, O., Corrado, G., Hughes, M., and Dean, J. Google's neural machine translation system: Bridging the gap between human and machine translation. *CoRR abs/1609.08144* (2016).

[69] Xiong, L., Xiong, C., Li, Y., Tang, K., Liu, J., Bennett, P. N., Ahmed, J., and Overwijk, A. Approximate nearest neighbor negative contrastive learning for dense text retrieval. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021* (2021), OpenReview.net.

[70] Yang, W., Zhang, H., and Lin, J. Simple applications of BERT for ad hoc document retrieval. *CoRR abs/1903.10972* (2019).

[71] Zhang, Y., Lo, D., Xia, X., and Sun, J. Multi-factor duplicate question detection in stack overflow. *J. Comput. Sci. Technol. 30*, 5 (2015), 981–997.