University of São Paulo
Institute of Mathematics and Statistics
Bachelor of Computer Science

DeeperMatcher

Using LLMs for crowd based requirements engineering

Arthur Pilone Maia da Silva

Final Essay

mac 499 — Capstone Project

Supervisor: Prof. Dr. Paulo Roberto Miranda Meirelles
Co-supervisor: Prof. Dr. Fabio Kon

During the development if this work,
the author received financial support by
CNPq (proc. 465446/2014-0) and
FAPESP (proc. 2024/00957-8 and 2014/50937-1)

São Paulo
2024

To the shy Luganese snowflake that first blessed
my warm face on a mid-November night and
teased my heart with the true size of the world.

Acknowledgements

Firstly, I thank my family for all the love and support that has helped me stay confident and pursue my goals, whatever they were. Besides supporting me, my friends have also been essential by keeping me motivated and optimistic not only through this eventful year but throughout my whole life.

I am profoundly grateful for the support, advice, and guidance of Prof. Michele Lanza, who warmly welcomed and hosted me as a visiting student in Lugano. His insights were of great value to this project and will follow me throughout my career as a researcher. I am equally thankful for my lab colleagues, who gave me such good company and helped me feel at home thousands of kilometers from everyone I knew. I would also like to thank Prof. Walid Maalej for accepting our invitation to collaborate as we followed up on such an influential piece of his previous work.

Last but certainly not least, I am immensely grateful for my advisors, Prof. Paulo Meirelles and Prof. Fabio Kon, who've always put great trust in me and my work. I thank them deeply for investing in me, giving me ample space to grow, and gently persuading me into a life of curiosity.

Resumo

Arthur Pilone Maia da Silva. **DeeperMatcher:** ***Usando LLMs para engenharia de requisitos baseada na multidão***. Monografia (Bacharelado). Instituto de Matemática e Estatística, Universidade de São Paulo, São Paulo, 2024.

Times de desenvolvimento de aplicativos populares podem receber milhares de análises de usuários diariamente. Ao mesmo tempo, esses desenvolvedores usam canais de comunicação totalmente diferentes para conversarem entre si, tais como os issue trackers de seus repositórios de software. Embora acessíveis e facilmente administradas pelos desenvolvedores, o conteúdo das issues de desenvolvimento difere fortemente do que a maioria dos usuários escreve em suas análises. As issues podem não ter alguns dos passos necessários para reproduzir uma falha, ou alguma perspectiva que sustentaria a importância de uma nova funcionalidade, algo que pode estar presente em algumas das análises de usuário. Mais ainda, uma falha específica pode ter sido descrita em poucas análises de usuário e pode não chegar ao conhecimento dos desenvolvedores, sendo abandonada por anos na loja de aplicativos. Nós propomos uma abordagem para complementar issues de desenvolvimento com análises pertinentes, a fim de suprimir a disparidade entre o que os usuários escrevem em suas análises e o que os desenvolvedores mantém em suas issues. Nossa abordagem recupera automaticamente análises de usuário com alta similaridade textual semântica (STS) em relação ao conteúdo de uma dada issue, conforme calculada usando um grande modelo de linguagem (LLM), e sugere análises relevantes à compreensão da issue. Nós conduzimos uma avaliação preliminar usando dados dos projetos de software livre BikeSP e Brave, com mais de nove mil issues e 180 mil análises de usuários. Nossos resultados comprovam o potencial da abordagem. Além de elucidar o que os usuários pedem como funcionalidade ou reportam como falha, a informação presente nas sugestões pertinentes mostra o caminho para uma nova e promissora maneira de se aproveitar as análises de usuário para uma evolução consciente de aplicativos móveis.

Palavras-chave: Similaridade Textual Semântica. Issues de Desenvolvimento. Mineração de Repositórios de Software. Mineração de Retorno de Usuários. Processamento de Linguagem Natural.

Abstract

Arthur Pilone Maia da Silva. **DeeperMatcher:** ***Using LLMs for crowd based requirements engineering***. Capstone Project Report (Bachelor). Institute of Mathematics and Statistics, University of São Paulo, São Paulo, 2024.

Development teams of popular mobile apps can receive thousands of user reviews daily. At the same time, these developers communicate with each other using completely different communication channels, such as their software repository issue tracker. Although accessible and manageable for developers, the nature of the content in development issues differs starkly from what most users write about in their reviews. Issues may lack the steps to reproduce a bug or insights justifying the priority of a new feature request, which might be latently present in user reviews. Moreover, a specific bug might be reported by only a handful of user reviews and might go completely unnoticed for years abandoned in the app store. We propose an approach to augment software repository issues with pertinent user reviews to bridge the gap between what users write about in their app reviews and the issues managed by developers. Our approach automatically retrieves user reviews with high semantic textual similarity (STS) to the issue content, computed using a state-of-the-art large language model (LLM), and suggests reviews relevant to the developers' grasp on the issue. We conducted a preliminary evaluation using data from the Free/Libre/Open Source Software projects BikeSP and Brave browser, with over 9k issues and more than 180k user comments. Our early results testify to the potential of the approach. Besides providing insights into what users ask as a feature or report as a bug, the information found in the pertinent matches points toward a novel and promising way of leveraging user reviews for a concerted evolution of apps.

Keywords: Semantic Textual Similarity. Development Issues. Software Repository Mining. User Feedback Mining. Natural Language Processing.

Abbreviations List

SE	Software Engineering
RE	Requirements Engineering
CrowdRE	Crowd Based Requirements Engineering
STS	Semantic Textual Similarity
LLM	Large Language Model

List of Figures

[1] Source: https://tinyurl.com/kaban-board

Contents

Introduction

As computers became more powerful and more numerous in the second half of the 20th century, it became evident that the human effort necessary to comprehend and maintain the increasingly more extensive software programs grew exponentially with the size of the system. The ever-growing number of processes and systems that depended on large software products brought along more and more examples of the challenges associated with organizing the components of a complex program and the possible consequences that emerge when flawed programs cross the path of people who barely know of the program's existence[2].

Eventually, the experts realized that there would need to be established values, methods, and practices to ensure (i) the usability, stability, and correct function of software systems; (ii) that the vision development teams, stakeholders, and end users have for the product being developed converge; (iii) the extensibility and maintainability of the software system, while still preserving items (i) and (ii). Correspondingly, before the end of the 1960s, and shortly after the creation of the first large-scale computational systems, the field of **Software Engineering** began to grow (Valente, 2020). Its main intention would be to secure these three points by employing processes and guidelines empirically consolidated by years of practice in other engineering fields.

However, the fundamentally subjective nature of what makes a software product adequate to its users poses challenges to approaches based on the principles of what we understand as "traditional software engineering". Hence, at the turn of the century, agile software development methods emerged as a countermovement inside software engineering, advocating for a more human and personal approach to formulating the values and practices that guide the development of software systems. Time has shown that the agile way of developing products is more consistent and reliable than using some approaches advised by "traditional software engineering" practitioners.

At the same time that the widespread adoption of the World Wide Web and the advance of smartphones and app stores increased dramatically the number of users a medium-scale software project might achieve, the connection with the end users, as preached by the agile methodology, became increasingly unfeasible.

Amidst millions of user reviews and comments on app stores and social media, prioritizing which comments should be studied by the development team as starting points for new requirements is challenging and demands an unreasonable amount of human

[2] See **https://en.wikipedia.org/wiki/Therac-25**

analysis. Previous works explore classifying, understanding, and using user feedback to support app development, but the sheer scale at play continues to present itself to software engineers as both a curse and a blessing. While the large volume of user input constitutes a potential source of many valuable insights rarely elicited by the experience of one specific user but recurring in a pool of millions of messages, the overwhelming amount of useless information poses a formidable challenge (Pagano and Maalej, 2013).

The prospect of tapping into this considerable amount of data embedded in the millions of user reviews and posts on social media has attracted significant attention from the software engineering research community (Groen *et al.*, 2017; Snijders *et al.*, 2014). This scheme has been studied before, especially using Machine Learning models and Natural Language Processing (NLP) (Santos *et al.*, 2019; Stanik *et al.*, 2019; Mekala *et al.*, 2021), with varying levels of accuracy. Regardless, researchers have yet to find a solution that accomplishes the task of using this data efficiently and reliably.

Haering *et al.*, 2021 proposed a mechanism called *DeepMatcher* that automatically matches problem reports in app reviews to bug reports under the developers' knowledge using large language models. The initial version was only a proof of concept that covered issues and reviews written in English. Although interesting, the mechanism proposed by the authors is far from usable and, by itself, brings little to no value to a development team. Still, it pointed towards a promising application of recent technology to manage an amount of user data that was previously untreatable.

Therefore, with the purpose of better understanding whether the technology used by Haering *et al.*, 2021 in *DeepMatcher* could assist developers in leveraging the vast amounts of data produced daily by the users of large applications, we decided to design, implement and test DeeperMatcher. Initially built on the tracks of what the original authors of *DeepMatcher* had in mind, our project is a FLOSS working tool and aims to be a decent match for teams responsible for managing large mobile applications.

The development process covered by this essay consists of two cycles, one focusing on studying the potential the approach has at retrieving issues relevant to a given review and another focusing on augmenting the information from a given issue with that of automatically identified reviews. We discuss the motivations that guided each development cycle and the modifications made to the workflow architecture. Additionally, we conduct a preliminary evaluation on each cycle and register the results obtained alongside the insights they brought to the design of the approach.

Using the data from the replication package of *DeepMatcher* and other publicly available large software projects (*i.e.,* Brave[3]), we analyze the behavior and reliability of DeeperMatcher using single-case mechanism experiments and observational case studies (Wieringa, 2014). Additionally, the code used in our analyses is available in the following replication package: https://doi.org/10.5281/zenodo.13773021, as well as in the following GitLab repository: https://gitlab.com/ArthurPilone/deepermatcher.

The structure of this essay is as follows: In Chapter 1 we explore the concepts related to software development and natural language processing necessary to comprehend the

[3] See **https://github.com/brave/brave-browser/**

context it finds itself in, while in Chapter 2 we explain how we use text embeddings to measure the similarity of issues and reviews. We dedicate Chapters 3 and 4 to detailing the motivations, implementation decisions and evaluation of our first and second development phases, respectively. We discuss our results, limitations, previous and future work in Chapter 5 before closing our essay on Chapter 6.

Chapter 1

Related Concepts

It is relevant to lay a common ground for the theoretical basis of this project to assist the understanding of the following chapters of this essay. Thus, we dedicate this chapter to introducing or reminding the reader of the origin or definition of some concepts related to the technologies we use and software development in general.

1.1 Software Development

Historically, we can see two clearly distinct software development philosophies: *traditional software engineering* and the more modern *agile software development*. In the following sections, we present some of the context, motivation, and values of both philosophies.

1.1.1 Traditional Software Engineering

One of the cornerstones of the context in which this work finds itself is the area of **Software Engineering** (SE). As mentioned in the Introduction, the rapidly growing relevance and importance of software projects showed that creating and maintaining large systems was not a trivial task. As such, in the last four decades of the millennium, the field of Software Engineering was born. The main reasoning behind it would be to manage the production of software as that of any complex physical product of human design, such as that for electrical appliances or buildings. Accordingly, the first preachers from the newly born field of study advocated for the use of methods and processes employed in production lines with high upfront costs, little to no space for errors, and eventual changes to the product's design.

Among the different areas in Software Engineering, **Requirements Engineering** (RE) has its primary focus on identifying and analyzing **requirements**. As defined in the IEEE, 1990 standard, a requirement is "*a condition or capability that must be met or possessed by a system or system component to satisfy a contract, standard, specification, or other formally imposed documents*". While an engine might have the specification of achieving at least 300 horsepower at max power, a computer system might have the requirement of supporting at least 2000 simultaneous users. Hence, in the same way that a

machine must be designed until the smallest of screws based on its desired specifications before its assembly commences, many thought that software systems should have their requirements collected and their complete architecture designed before the start of their implementation.

Although the notion of requirements as a set of specifications fits nicely for an industrial design, the fundamentally subjective nature of what makes a software product adequate to its users challenges the definition of requirements for complex systems. Unlike that of a concrete product, the resistance of a system can't be measured as a physical value, for instance. Furthermore, the experience that the users have while using a software system can vary dramatically depending on factors such as the features of the physical devices used to interface with the system, environmental variables such as brightness, sound levels, or internet connection in the place and time they access the system, and cognitive aspects, such as the users' speed at understanding the information given by the system, what previous knowledge they must have before using it, or the task they aim to achieve while using it.

1.1.2 Modern Software Development

As the turn of the century approached, the software engineering community slowly started to conceive that the original approach proposed in the previous decades needed significant adaptations to challenge the change of pace brought by the advent of the World Wide Web. The rapid surge of new applications for complex systems with increasing numbers of users only made the gap between the realities of software engineering and traditional industrial design more apparent.

Agile Software Development

Ultimately, **Agile software development** methods emerged as a countermovement inside software engineering, advocating for a more human and personal approach to formulating the values and practices that guided the development of software systems.

The newly-born philosophy sprung from an attempt to devise a lightweight set of values and methodologies to guide software development better than the original practices suggested by the traditional software engineering approach. Such philosophy culminated in the creation of the Agile Manifesto[1] in 2001. Quickly gaining thousands of signatories worldwide, the manifesto preached twelve core principles, including:

> Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.
>
> Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.
>
> [...]
>
> The best architectures, requirements, and designs emerge from self-organizing teams.

[1] See **https://agilemanifesto.org/**

Following these and other core principles, Agile development comprises a set of methods and practices for software design, implementation, and maintenance (Beck and Andres, 2004; Julian *et al.*, 2019).

Tasks and Issues

The most essential of the agile practices is separating the development period into short cycles, called *sprints*, each with a set collection of atomic tasks the team should accomplish before its end. The set of tasks to be done by a team is known as its *backlog*, and their status can be tracked by their position in a ***kanban*** board. In physical contexts, such as the one show in Figure 1.1, these tasks are often associated with artifacts such as *post-it*s organized in a whiteboard based on their completeness or statuses.

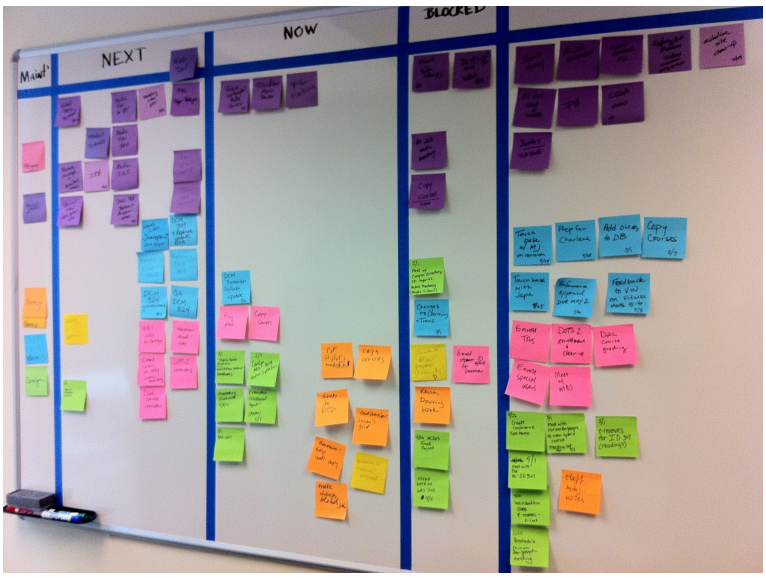

Figure 1.1: *Example of a Kanban board used by a development team.*[a]

[a] Source: https://tinyurl.com/kaban-board

Another alternative to represent these tasks is through the use of **issues** stored in specific tools known as **issue trackers**, which intend to mimic the physical workflow of organizing tasks on a *kanban* board. As an agile team goes through the requirements of their product, they dilute each requirement into a set of tasks (or issues) the developers will tackle during the subsequent sprints. While the use of whiteboards brings a physical dimension to the management of issues, most teams of large-scale projects depend on specific issue trackers to manage their requirements. Besides efficiently storing thousands of issues at a time, these tools usually support labeling issues, assigning them to a developer or linking them to another artifact.

Arguably more important than storing and organizing the issues and tasks to be done is actually persisting and centralizing the code the developers write. We name as **code repository** the platform that centralizes all the code developed by a team and every one of its incremental changes. In the late 2000s and early 2010s two companies, GitHub[2] and

[2] See **https://github.com/**

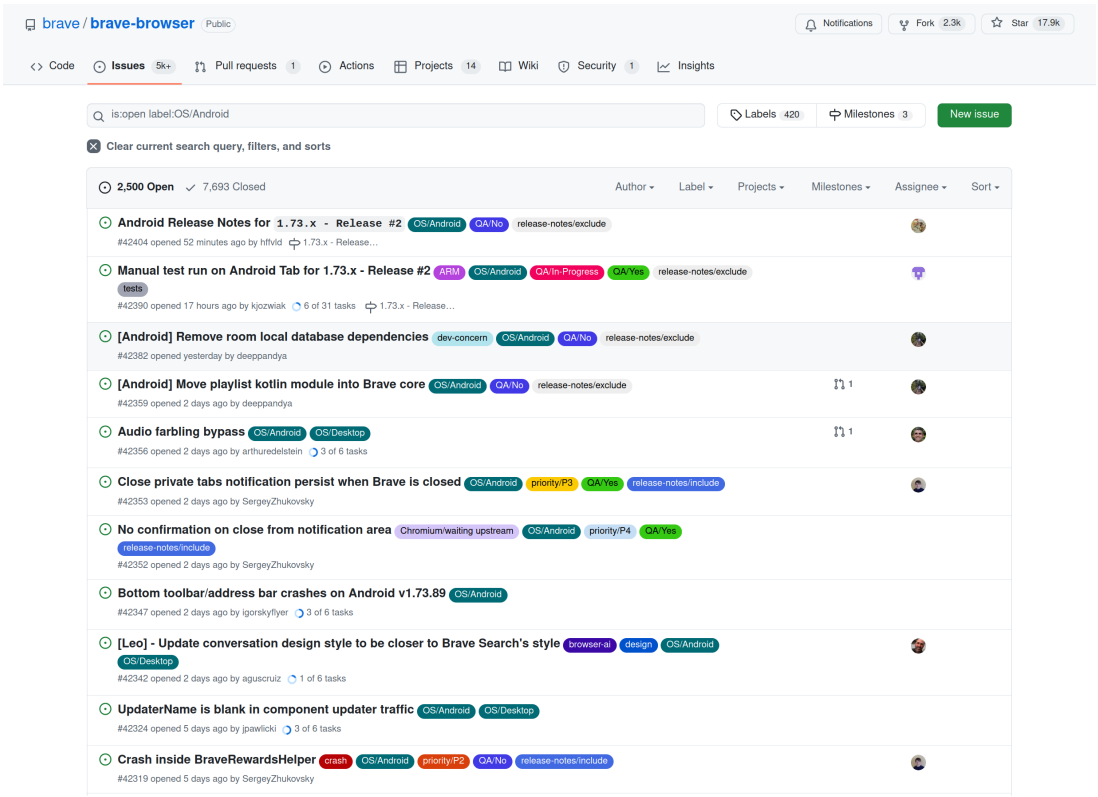

Figure 1.2: *GitHub issue tracker used by the development team of the Brave browser.*

GitLab[3] appeared serving individuals and companies alike the possibility to host their software repositories for free, opening a new age for collaborative, free and open-sourced software development.

As the most immediate use of issues is to track the implementation of a given piece of code which satisfies (at least partly) a requirement, there exists a natural dependency between the issue trackers and the code developed by the team. As such, most issue trackers are integrated with the project's repository. With this integration a developer might, for instance, submit a new change to the repository in such a way that the progress of an issue is automatically updated in the issue tracker, considerably decreasing the overhead of manually updating issue statuses. This smaller overhead, in turn, helps keep the issue tracker updated accordingly to the project development and boosts the function of the tracker as a hub for visualizing and organizing the project's progress.

Moreover, some issue trackers even support the integration of *service desks*[4], platforms in which users might open a *ticket* to contact the developers. This allows developers to see and treat the user ticket as an issue, which eases its integration into as a new task in their existing workflow.

[3] See **https://about.gitlab.com/company/**

[4] See **https://docs.gitlab.com/ee/user/project/service_desk/**

Integrating the Users

Another big pillar of the agile methodology is the search for constant and immediate user feedback throughout the product's development. If the team develops a software product for a specific client, it would be ideal to have a meeting with the client at least once at the end of every sprint to receive feedback on the tasks competed and requirements implemented in that sprint, so to easily adjust the planning of the following steps if changes are necessary.

Accordingly, among the practices that summarize the twelve core agile principles is the habit of writing requirements in the form of **user stories** (Cohn, 2004). In summary, each requirement is registered in the form of a small anecdote written from the point of view of a hypothetical target user describing what is the expected behavior of the system, how they would interact with it and to what end. This forces the developers to keep in mind what aspects of the requirement are most important to the users and how it improves their experience with the software product.

Although valuable for development teams and still considered effective for capturing user requirements, writing user stories may not be feasible for every situation and every type of requirement. Moreover, having a skilled customer or a product owner write good user stories is often far from reality for many software projects. It is thus common to see teams choosing not to adopt user stories and just keep track of their requirements as tasks to be implemented and issues to be resolved (Montgomery *et al.*, 2024).

However, the convenience of focusing on recording development and maintenance tasks comes with the risk that the development tasks and the actual needs of the users may drift apart. This scenario may be even more challenging for smartphone apps with large user bases (Pagano and Maalej, 2013). A certain requirement related to a specific group may never get to the development team due to the absence of direct communication between developers and users. It is not hard to see how the problem is exacerbated when it comes to bug reporting. The vast range of operating systems, device models, execution environments, and user contexts make the task of accurately understanding how a given software product should behave in every possible condition nearly unattainable (Martens and Maalej, 2019; Maalej, Biryuk, *et al.*, 2024).

With millions of user reviews and comments on app stores and social media, prioritizing which comments should be analyzed by the development team to capture potential requirements is challenging and demands an unreasonable amount of manual effort. It is still unclear whether there could be a possible way to reliably use this large amount of data instilled in reviews and posts on social media for supporting software and requirements engineering tasks. This prospect is currently under attention of the SE and RE research communities, and has received different names, such as *Data Driven Requirements Engineering* (Maalej, Nayebi, *et al.*, 2016; S. Lim *et al.*, 2021) and **Crowd-Based Requirements Engineering** (CrowdRE) (Groen *et al.*, 2017; Snijders *et al.*, 2014).

The main reason why automatically treating large amounts of feedback has such a high complexity is strictly linked to the fundamental context which it is based on — processing natural language.

1.2 Processing Natural Language

Natural language processing (NLP) deals with programmatically storing, ordering, retrieving, and comparing fragments of text written in languages naturally used by humans, such as English, Italian, Portuguese etc.. Perhaps the most recurring task involved in NLP is retrieving documents related to a query string. Popular use cases include retrieving web pages corresponding to a set of search terms, wiki pages covering a given concept, or publications that satisfy a given query string. That is, search engines in general.

A common step for most (NLP) tasks is to split the input text into **tokens**. The length of a token can vary significantly depending on the use case and technology used, but can be thought to roughly correspond to a single word of the original text.

A similar task that intersects with document retrieval is **semantic search**. It deals with retrieving words, sentences, or documents that not only include terms in a given query string but also synonyms and other expressions that convey a similar meaning. By focusing on the semantic value of the text fragments, that is, what they mean, semantic search presents itself as a more delicate and complex task than simple term-based fragment retrieval. While semantic search is still a task receiving constant attention on the lookout for better approaches, the use of Term Frequency – Inverse Document Frequency (TF-IDF) (Sammut and Webb, 2010) has dominated term-based retrieval tasks for decades.

One of the alternatives to perform semantic search is through the computation of a **semantic textual similarity** (**STS**) value. This measure intends to capture how close the meanings of two text fragments are.

Still to this day some NLP tasks, especially STS computation, have no definitive solution. The unmanageable number of different combinations of terms that convey similar meanings makes the field inheritably complex for a programmatical and discretely bound science. Furthermore, the inherently subjective nature of communication and the varying clarity of human-written text make tackling some NLP tasks with algorithmic approaches almost impossible. However, recent advances in the field of artificial intelligence (*i.e.,* the surge of large language models) have started to bring these tasks to the realm of feasibility.

1.3 Artificial Intelligence and Large Language Models

Initially created in the pursuit of replicating the inner workings of a brain, artificial intelligence (AI) models differ starkly from traditional algorithms. The behavior of an artificial intelligence model is not determined solely by a set of instructions, but by a dataset used to train it — that is, pin down its many parameters. After a model is trained, the output it delivers to a given input roughly boils down to a set of (mostly obscure) mathematical computations dependant of the parameters found during training and, for some architectures, a non-deterministic or stochastic component.

As some NLP tasks have recurringly presented themselves as challenging for traditional algorithmic approaches, artificial intelligence has repeatedly been considered a possible

alternative to solve them. However, the mathematically based and data-driven inner workings of AI models force them to translate whatever text fragment they process into a mathematical representation. As such, the input text is broken into tokens, and each token is mathematically represented as a number or list of numbers. We can condense this numeric representation to an **embedding**, a list of floating point numbers that a model uses to store semantic information relative to a token.

Manually designing a mathematical abstraction capable of efficiently representing the immense plethora of different meanings any given word or sentence might have is far from feasible. Accordingly, only extensive and complex models come close to being reliable for the most delicate NLP tasks. These models usually have hundreds of millions or even billions of parameters and can only be trained by large organizations with plenty of resources. Recently, they have been categorized under the name of **large language models (LLMs)**

Typical LLMs are based on the Transformer architecture (Vaswani *et al.*, 2017), which promotes the exchange of information between embeddings of tokens from a single string. This interaction is heavily connected to the concept of *attention* (Kim *et al.*, 2017) and is the basic idea behind most recent advances in NLP. As the models process text, the embedding of a token is influenced by the embeddings of its neighboring tokens, similar to how the meaning of a noun is influenced by the adjectives surrounding it. At the encoder output, each embedding carries information that reflects the token context. These final embeddings can be called **contextualized embeddings**. Every LLM has a maximum number of tokens a single input can have, which we call its **context size**.

Chapter 2

Using Text Embeddings

As the goal of the approach is to fuse the knowledge embedded in issue trackers with that found in user reviews, its core mechanic trivially lies in finding pairs of issues and reviews that concern the same bug, feature, or context in general. Development teams that receive large amounts of user feedback should be able to use our approach to automatically identify what issues in their issue tracker a user review may correspond to and vice-versa. We can understand this mechanic as a search of pairs with high semantic textual similarity (STS).[1]

As seen in Section 1.3, LLMs naturally produce embeddings as they process and encode text. These embeddings have a key property that is fundamental to our goal, and imagining them as vectors in an arbitrary vector space is of great help in perceiving it. As they strongly correlate with the meanings of each token (Jiao and Zhang, 2021; Ethayarajh *et al.*, 2019; Mikolov *et al.*, 2013), we can try to **measure the semantic similarity of two texts by the proximity between their embeddings in their high-dimensional vector space**. This insight bases the *DeepMatcher* proof of concept of Haering *et al.*, 2021 and allows us to measure the STS of two text fragments by using the distance between their contextualized embeddings. We illustrate the essential steps to computing the STS value in Figure 2.1

Therefore, the system we propose uses the contextualized embeddings created by LLMs to measure the similarity of user reviews and development issues and suggest matching pairs that point to the same topic. Since the metric used for computing the distance between embeddings is not a crucial factor of the approach, it is subject to change, and the system should support different similarity thresholds. However, our implementation follows the original *DeepMatcher* proof of concept and uses cosine similarity[2].

Before presenting the matches with the highest similarity to each issue or review (*i.e., relevant matches*), our approach uses an arbitrary similarity threshold (σ) to filter only *pertinent matches*. Non-pertinent high-ranking matches contain reviews only partly related to the issue and other false positives. The approach retrieves the most pertinent

[1] See Section 1.2

[2] See **https://en.wikipedia.org/wiki/Cosine_similarity**

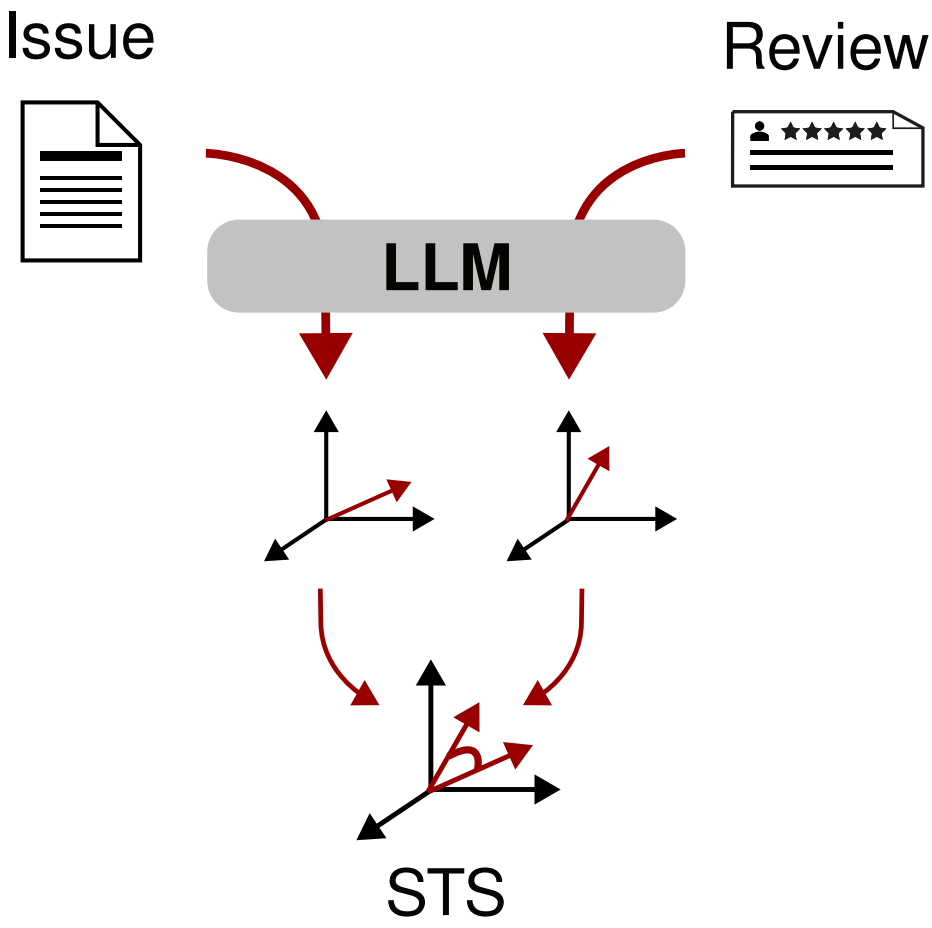

Figure 2.1: *Mechanism used to measure the STS between issue and reviews.*

matches for each issue or review and presents them to the developer.

It is noteworthy that not all pertinent matches contain enough new information to enhance comprehension of the corresponding pair. Accordingly, we define as *useful matches* those that are pertinent to the issue or review it corresponds to and augment it by eliciting new insights or providing additional information.

A guiding principle of our approach is to support the requirements engineering process while maintaining a human in the loop (Andersen and Maalej, 2024). While leveraging LLMs, it is still important that a competent developer reviews the suggestions given by our workflow to mitigate the risk of fully relying on the inherently imperfect nature of artificial intelligence. Implementations and future extensions to our architecture should be cautious to avoid introducing imprecise results that could jeopardize the development of software products and their users.

Chapter 3

Matching Multilingual Reviews

For the first development cycle of the workflow, we focused on implementing a mechanism to retrieve issues similar to a single review. The workflow could also repeat the process for a collection of reviews saved in a specific format. We use this chapter to bring up the motivations for this use-case of the tool, the decisions we made implementing it, our preliminary evaluation, and its results.

3.1 Motivation

In addition to the challenges brought by the overwhelming number of reviews, the language disparity between user reviews and the technical writing of developers poses a formidable hurdle for CrowdRE. Apart from the differences in the language used, it is common to see critical information missing in the user reports, such as app version or hardware model (Martens and Maalej, 2019; Zimmermann *et al.*, 2010). An approach that automatically matches reviews and issues could aid developers seeking to understand better the problems and requests users describe in their reviews.

Being the first to propose using text embeddings to match user reviews with issues, the work of Haering *et al.*, 2021 undoubtedly serves as a cornerstone for possible new applications of LLMs for requirements and software engineering. Besides its novelty and relevance, their proof of concept achieved a noteworthy precision that merits the approach with further study.

Although intriguing, the prospect of matching reviews to issues, by itself, brings little value to the software development process. Only a complete workflow built on top of the proof of concept of *DeepMatcher* could leverage its strengths to support software evolution and maintenance. An extensive software system is necessary to involve the artifact matching mechanism with the steps of data collection, treatment, storage, and presentation.

When a complete and reliable workflow matches a user feedback item to a bug report from the issue tracker, the developers should be confident that the user is reporting something they already know exists. Moreover, the issues matched to a bug report should

help the developers identify which might be the probable cause of the problem the user is reporting.

Far from being a negative result, the absence of pertinent matches should be correctly treated by the approach. Relevant reviews with no matching issue could point to a latent bug in the case of a crash report, for instance, or a suggestion previously unheard of, in the case of a request. When the approach finds no matching issue for a group of reviews, it should present them to the developers. Then, they might decide whether a new bug or feature request is latently present in the reviews.

3.2 Implementation

Figure 3.1 depicts the main components of our approach that matches issues to a given review. A guiding principle in our design is to give developers the flexibility to switch and modify components that are likely to change due to rapidly evolving NLP techniques or specific team preferences.

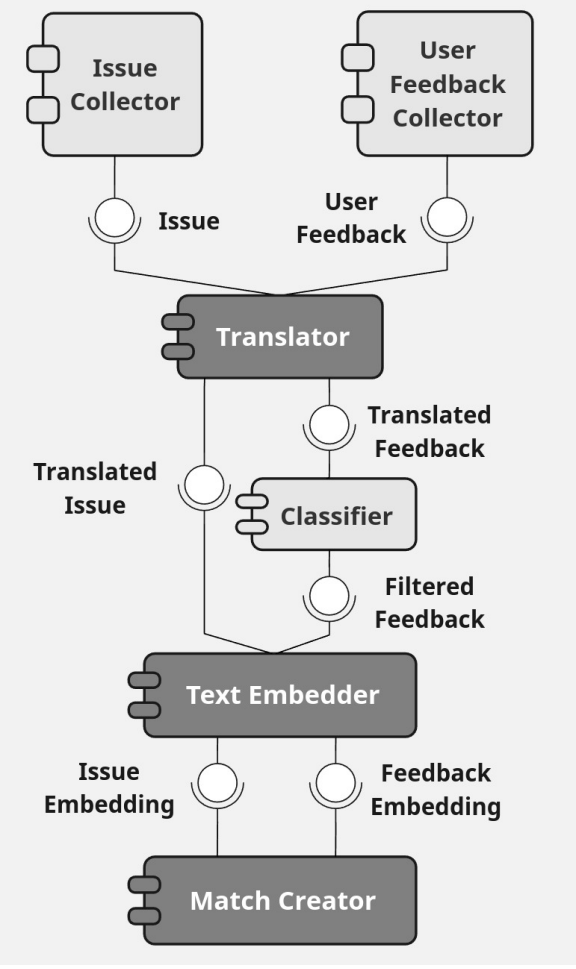

Figure 3.1: *Fundamental components of the approach for matching issues to a given review.*

As we're proposing a complete workflow surrounding the suggestion of pertinent matches, our approach wouldn't be complete if it didn't involve collecting the required

data. Therefore, the first components of the workflow are responsible for collecting the issues and reviews related to the desired app. These components interact with external APIs and crawlers to gather issue and review data. It is a high priority that they follow a straightforward interface, ensuring that future maintainers can easily implement classes to extract data from new sources.

For the first development cycle, we included an issue collector for public GitLab repositories besides stubs for other crawlers and collectors inherited and adapted from the *DeepMatcher* replication package. We initially chose GitLab due to its widespread adoption for hosting FLOSS projects and because it hosts the sample project used for our preliminary evaluation. We already anticipated implementing extensions for these components in the near future, as would eventually happen for the second development phase.

In general, the classes used for manipulating issues and reviews should also be agnostic to the specific project analyzed to ensure high generalizability. The choice of the repository or issue-tracking system developers use should not directly affect the workflow behavior. It is equally desirable that the app store used to publish the app is unimportant to the remainder of the approach. We could also extrapolate the notion of an app review to work with generic 'user comments' regardless of where the user feedback comes from.

As for this phase we were interested in supporting artifacts in languages other than English, we implemented a simple text-translator component using the `googletrans` API, which is built on Google Translate. This component can translate text between any two languages supported by Google's API and was used to translate issues and reviews to English. The issues were translated during collection and user reviews after they are entered in the command-line interface.

Before embedding user reviews, a team of developers may choose to filter those that might be irrelevant (Stanik *et al.*, 2019) or do not correspond to any of the issue types maintained in the repository. For instance, the team might not be interested in reviews requesting new features or simply praising the app. Accordingly, our first implementation included a component for classifying user feedback (Maalej, Biryuk, *et al.*, 2024). We used the classifier from Stanik *et al.* (2019) to allow classifying user reviews into three classes: "Irrelevant", "Feature Request", or "Bug Report". Using their replication package, we trained a simple classifier built on the DistilBERT (Sanh *et al.*, 2019) model from the `Transformers` Python library.

The text embedder is the part of the system responsible for receiving a text fragment (i.e, a review or an issue) and returning a single embedding used to compare it to other text fragments. Following the approach described in Chapter 2, we provide the text to an LLM and use the contextualized embeddings it computes to generate a single embedding for the given string. The system creates an embedding for each review and developer issue. During this first development phase, we follow the findings from Haering *et al.*, 2021 and use only the issue titles for textual embedding, as they have been shown to adequately summarize the issue content. As this component relies heavily on the specific LLM used, our architecture also allows for changing it according to the project's needs.

During this cycle, we implemented two text embedding approaches, the first mirroring that of *DeepMatcher* and illustrated in Figure 3.2. We feed the input text to a DistilBERT

model, which tokenizes it and generates a numeric embedding for each token. The model adjusts each token's embedding based on the embeddings of neighboring tokens, so that every embedding also carries information about the context in which the token appears. Next, we use the SpaCy (Honnibal *et al.*, 2020) part-of-speech (POS) tagger to identify tokens related to nouns and use the `pytokenizations` module to align the tokens from the SpaCy and DistilBERT models. Finally, we collect the contextualized embeddings generated by DistilBERT for the tokens that the SpaCy model identified as nouns. To create a single embedding for the entire text, we compute the average of all embeddings after the noun filtering.

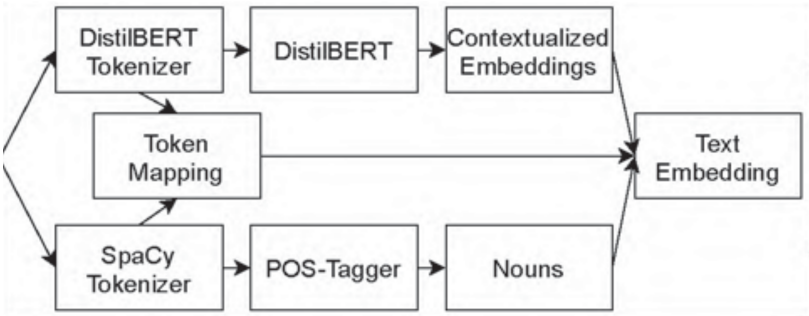

Figure 3.2: *Overview of the embedding process of* DeepMatcher. *Adapted from (Haering et al., 2021).*

The second embedding approach uses the `SentenceTransformers` (Reimers and Gurevych, 2019) Python library. The model, "`all-MiniLM-L6-v2`," and the library are both designed to compute a single meaningful embedding for a given text string, thus providing an equivalent approach to the text embedding process used in *DeepMatcher*.

After the embeddings have been computed, we use the cosine similarity to quantify the similarity between issue and review embeddings. The system presents the user with the issues with the titles most similar to the given review. The user may specify a similarity threshold, and the number of matches DeeperMatcher should suggest for each user review.

3.3 Evaluation

Our evaluation for the first development cycle investigated how well the implementation worked and identified potential areas for improvement. We focused on answering the following question:

- How reliable are the matches suggested by DeeperMatcher for English and other languages?

DeeperMatcher builds upon the *DeepMatcher* proof of concept. Therefore, it is expected that DeeperMatcher functions properly with the issues and reviews used to validate the original proof. A fundamental principle in the development of DeeperMatcher was to ensure its applicability to data compatible with *DeepMatcher*.

Using the command-line interface of DeeperMatcher, it was easy to verify that our

architecture produces matches equivalent to those from the *DeepMatcher* proof of concept. Figure 3.3 illustrates an example where DeeperMatcher results coincide with those from *DeepMatcher*. We anticipate that the text embedding generation in DeeperMatcher will improve and be easily adjustable. Therefore, our evaluation did not focus on the direct characteristics of the numeric values created for the text embeddings. Instead, we focused on assessing the reliability of DeeperMatcher as a tool.

```
--------------------------------------------------------------------------------
Review: 'So many bugs... Plays in background, but no controls in notifications.
When you tap the app to bring up the controls, the video is a still screen. Navi
gating is a pain. Resuming forgets my place constantly. Basically unusable'
--------------------------------------------------------------------------------
Similarity: 0.857
Issue text: '[Android] On video playing the navigation bar is not hidden on some
 tablets'
--------------------------------------------------------------------------------
Similarity: 0.856
Issue text: 'android navigation bar, shown after a click, shifts and resizes ful
l-screen video'
--------------------------------------------------------------------------------
Similarity: 0.849
Issue text: 'Play/pause button icon is not shifting while pausing the audio on n
otification area'
--------------------------------------------------------------------------------
```

Figure 3.3: *Screenshot of matches identified by* DeeperMatcher *when prompted with a review from Table III of* DeepMatcher

3.3.1 Data Acquisition

To compare the performance of DeeperMatcher with its predecessor, we used the same data that was used to evaluate *DeepMatcher*. This allows us to observe changes in the matches suggested by both systems. We utilized data from Haering *et al.* (2021), which includes English-written issues and reviews from two large-scale FLOSS projects: the VLC media player and the Signal messaging app. We selected data from these projects for our comparison.

Additionally, to evaluate the reliability of DeeperMatcher with data in languages other than English, we collected 574 issues and 69 user reviews in Brazilian Portuguese from the medium-scale project BikeSP (Pilone *et al.*, 2024). The BikeSP app developers manually associated each review with the corresponding issue when the user comment referred to something identifiable solely by its text. Out of the 69 reviews, only 23 had a corresponding development issue. The replication package referred to in the introduction[1] also includes the reviews and issues from this project.

3.3.2 Evaluation Methodology

First, we tested whether the text embedder implementation derived from *DeepMatcher* produced results equivalent to those of the original prototype. We provided 100 randomly

[1] See **https://doi.org/10.5281/zenodo.13773021**

selected user reviews from the Signal Messenger app data to DeeperMatcher and verified if the three issues suggested by the tool matched those listed by *DeepMatcher* for the same input.

Next, we focused on the main part of our evaluation. We conducted a single-case mechanism experiment (Wieringa, 2014) to study how DeeperMatcher performed on data from the BikeSP and VLC projects and how variations in input affected the results. Using the command-line interface (CLI) of DeeperMatcher, we set the source of issues to the repository for the desired project and instructed the tool to translate user feedback from Brazilian Portuguese to English. We also disabled the minimum similarity threshold and configured the system to provide five candidates for matching issues for each review.

We entered the reviews one by one and analyzed the results individually. For reviews where the Brazilian developers identified a matching issue, we counted how frequently DeeperMatcher listed the issue the developers had in mind. This procedure was repeated twice for each user review from the Brazilian project: first using the text embedding method from *DeepMatcher* and then using our newer text embedding method based on `SentenceTransformers`.

Finally, we conducted a qualitative analysis by selecting issues from the VLC and BikeSP projects and comparing the results using newly created reviews. We observed how the matches suggested by DeeperMatcher varied with different text embedding methods.

3.4 Results

As mentioned in Section 3.3, the evaluation for the first development cycle was split into two experiments: one quantitative and another qualitative.

3.4.1 Quantitative Results

Our test suite confirmed that our implementation was consistent with the original proof of concept from which it was derived. For all 100 reviews sampled from the Signal app data, DeeperMatcher consistently suggested the same three issues as *DeepMatcher*.

However, when testing the text embedding method derived from *DeepMatcher* with the BikeSP dataset, DeeperMatcher struggled with most reviews. Out of 23 reviews where the Brazilian developers had identified a matching issue, only three (13.0%) were included by DeeperMatcher in its list of the five most similar issues. In these cases, the mean similarity value for the correct issue was 81%, with two issues listed as the 5th most similar and one as the 2nd most relevant.

In contrast, results improved significantly with the newer embedding method, as seen in Figure 3.4. Using the LLM from the `SentenceTransformers` library, DeeperMatcher correctly suggested the issue for 13 out of 23 (56.5%) reviews. Notably, most of these matches had a similarity value of less than 80%, indicating effective but modest alignment.

These results demonstrate that the accuracy of matches was highly dependent on the text embedding method used. They also suggest that further improvements to DeeperMatcher were needed to achieve higher prediction reliability.

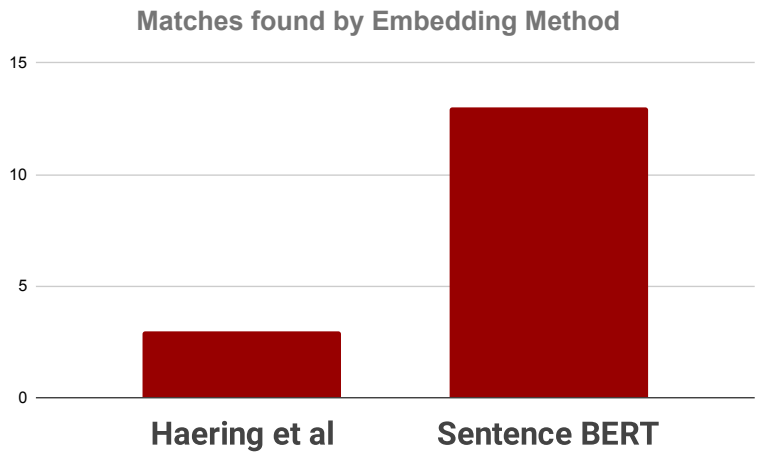

Figure 3.4: *Number of ground-truth matches found by the two embedding methods.*

3.4.2 Qualitative Results

We noted and investigated some interesting patterns as we experimented with the embedding method inspired by *DeepMatcher*. Although many suggestions did not correspond exactly to the issue representing the implementation of a new feature, they often included issues describing fixes or extensions related to the feature. For instance, as illustrated in Figure 3.5, DeeperMatcher did not identify the issue describing the creation of a new screen for the app. However, it suggested another issue with high similarity (86%) that described a problem with the screen.

```
--------------------------------------------------------------------------------
Review: 'Feedback das contestacoes agrupados em uma só tela'
--------------------------------------------------------------------------------
Similarity: 0.860
Issue text: 'Contestações croppadas na tela listar contestações'
--------------------------------------------------------------------------------
Similarity: 0.835
Issue text: 'Adicione informações sobre pausas e contestações às telas informati
vas'
--------------------------------------------------------------------------------
Similarity: 0.792
Issue text: 'Script que facilita gerenciamento de contestações'
--------------------------------------------------------------------------------
Similarity: 0.769
Issue text: 'Permitir que contestações sejam reprovadas com um dado motivo'
--------------------------------------------------------------------------------
Similarity: 0.758
Issue text: 'Telas com perguntas a serem respondidas pelos usuários'
--------------------------------------------------------------------------------
```

Figure 3.5: *Suggested issues for a review requesting a new app screen: Although the issue related to the creation of the new screen is not listed, the first suggested match is a fix for a problem with the existing screen.*

Furthermore, we observed that the original text embedding method performed poorly when user reviews were *considerably longer than the corresponding issue summary*. The ad-

ditional text created noise that interfered with its contextualized embedding and increased the distance to the embedding of the related issue.

To analyze the relationship between feedback length and accuracy of the match, we experimented with forging user reviews of different lengths describing requests and reports for problems present in the issue database. For the issue "Impedir início de viagem se economia de bateria estiver ligada" ["Stop users from starting a trip if the battery saver is on"], the review "Não existe uma maneira de impedir que eu inicie uma viagem se eu estiver com economia de bateria?" ["Isn't there a way to stop me from starting a trip if I have the battery saver on?"] led to match a with an 83.1% similarity score, but "Por que não me impedem de iniciar uma viagem se sabem que eu estou com economia de bateria?" ["Why don't they stop me from starting a trip if they know I have the battery saver on?"] did not lead to a satisfying match.

Surprisingly, this problem was not exclusive to translated issues. In the VLC dataset, we observed that the issue "Audio cuts off on Android" was the third suggestion DeeperMatcher provided when prompted with the review "The audio keeps cutting off," with an 80% similarity rating. However, altering the review text to "I don't understand why the audio keeps getting cut off" caused DeeperMatcher not to list the corresponding issue among the first ten matches.

Repeating these two tests with the newer embedding method, we observed signs of improvement. Although the examples in Portuguese yielded the same results for the `SentenceTransformers` embedder, we were unable to create a user review containing both "audio" and "cuts off" without DeeperMatcher suggesting "Audio cuts off on Android" as a matching issue.

Our evaluation showed that DeeperMatcher **was still unreliable** at the end of the first development cycle. Further improvements were necessary before the tool could be applied in practice. It appears that the older text embedding approach performed poorly with the Brazilian Portuguese data, failing to identify the relevant issues manually identified by the developers. Additionally, we identified a new potential point of failure in how DeeperMatcher handles longer user reviews.

3.5 Preliminary Discussion

From the preliminary evaluation of our first development cycle we realized that several possible improvements to DeeperMatcher should be explored before it could deliver reliable matches for any project in any language. Our results were somewhat less favorable than those reported by Haering *et al.* (2021). However, this does not contradict the findings from previous work, nor does it necessarily indicate degraded performance with the new system architecture. Nevertheless, we reflected on our evaluation results to identify potential **improvement points** for DeeperMatcher and its architecture.

First, our methodology differed from that used to evaluate the *DeepMatcher* proof of concept. While its authors focused primarily on identifying matches with bug reports, we expanded the scope to include reviews and issues related to new features. Additionally, our evaluation method diverged critically from *DeepMatcher*'s approach. By having access

to the development team, we collected precise information on which issues were created in response to each user review. This valuable data allowed us to conduct a more rigorous evaluation. Instead of merely counting matches considered relevant by coders, we quantified exact matches identified by individuals actively involved in the project.

Additionally, we reflected on the differences between the new data we used and the datasets used for the *DeepMatcher* evaluation. Setting aside the language aspect, the project analyzed in our study was critically distinct from those in the DeepMatcher evaluation. Although the number of issues from the Bike SP project was comparable to that of the VLC project, the issues from the Brazilian team included not only feature requests and bug reports but also management tasks. Moreover, these issues were frequently interrelated, as features implemented by the team were often expanded or require new fixes. This high interconnectivity increased the similarity among different issues in the repository, making it challenging for the review embeddings to be distinctly closer to a single issue.

Therefore, we argue that DeeperMatcher can exhibit variable performance depending on the issue repository. Implementing additional pre-processing or pre-selection steps between issue collection and its use for suggestions may be crucial for achieving more accurate matches.

Improvement 1
It might be necessary to **cluster issues based on the common features** they refer to or to **exclude issues that might be excessively technical or irrelevant** to what users can review. This step could be achieved through manual filtering by the development team (*e.g.,* by adding an additional issue field) or by integrating a new dedicated component into the architecture of DeeperMatcher.

When checking the influence of review length on the resulting text embedding, we observe a limitation inherent to contextualized embeddings and LLMs in general. As newer and more powerful LLMs are developed, their high-dimensional textual embeddings tend to improve the matching metrics significantly. The improvement in results after we changed the text embedding highlights how more recent embedders can mitigate this problem. Therefore, we conclude that constantly updating the models used for the text embedding process should enhance the matching performance.

Improvement 2
Switching DistilBERT for the original BERT or a larger model like Meta's Llama3 should be a straightforward upgrade due to our proposed architecture. Additionally, leveraging our system's adaptability to **include LLMs specifically built for clustering similar texts**, such as those from `SentenceTransformers`, can further improve the text embedding process.

Another way to mitigate the problem with imprecise embeddings in longer texts is to add a pre-processing step before the embedding process. The goal would be to select only

the most relevant parts of the review or issue text, thereby preemptively de-noising the text.

Improvement 3

Using another review processing component (Maalej, Biryuk, *et al.*, 2024), one might **delineate the most essential parts of each review**. Alternatively, a review processing component could **cluster reviews or issues** before DeeperMatcher searches for matches.

Unsurprisingly, translating the input text before creating the embedding negatively impacts the matching performance. The translation process can introduce slight changes in meaning, as different nuances may be ignored or lost. Feeding longer text entries to the translator increases the likelihood of these linguistic losses affecting the embedding. The added noise from translations also contributed to why DeeperMatcher struggled with the Brazilian Portuguese dataset. In addition to being in a different language, our reviews were, on average, longer than those in the DeepMatcher evaluation dataset, averaging 38 words per review compared to 33 in the original dataset (Haering *et al.*, 2021).

Improvement 4

By incorporating text embedding methods that work directly with the languages used by developers and their users, such as Brazilian Portuguese in our sample project, the need for a text translator component could be eliminated. Reducing this complexity might enhance the performance of DeeperMatcher.

When studying the results with our approach, clear evidence supports one of our core design choices: many identified improvements could be implemented without major restructuring of DeeperMatcher, thanks to the easy interoperability of its fundamental components, particularly the text translator and the text embedder.

Chapter 4

Automatically Augmenting Issues

Although the evaluation results for the first development phase were less than ideal, they still showed that the approach was improving on top of the proof of concept of Haering *et al.*, 2021. By including issues and reviews in multiple languages in the DeeperMatcher's domain, we dramatically increased the scope in which we tested the mechanism. However, the results of the rest of the evaluation revealed that there was plenty of room for improvement in the monolingual application of the approach.

Taking a step back, we realized there was a completely different use case for the mechanism we were developing, yet not so technically distant from what the implementation was doing — retrieving reviews relevant to a given issue instead of searching for issues similar to a given review. Even restricting ourselves to only using issues and reviews in English, our new use for the workflow would be a significant contribution on top of the original *DeepMatcher* proof of concept.

4.1 Motivation

Although accessible and manageable for developers, the nature of the content in their issue trackers differs starkly from what most users write about in their reviews. Issues may lack the steps to reproduce a bug or insights justifying the priority of a new feature request, which might be barely recurring in user reviews. Moreover, a specific bug might be reported by only a handful of user reviews and might go completely unnoticed for years abandoned in the app store.

To enhance the information available to developers with the users' perspective, some teams openly expose users to their backlogs by integrating service desks into their workflows. Other teams encourage users to file inquiries directly on their issue trackers, often with a high technical entry barrier. The biggest disadvantage of these solutions is that they ignore the immense amount of reviews that users are often willing to collectively provide for a popular app in the form of tweets or reviews, which are commonly left to be forsaken.

Once *pertinent* reviews are promptly available to developers, they might augment the issue they are working on with details it might lack. For example, for a bug-fixing

task, reviews might provide additional information for a specific software and hardware configuration (Zimmermann *et al.*, 2010). Similarly, a feature request (Herzig *et al.*, 2013) could benefit from insights directly from the intended audience.

When the approach matches highly pertinent reviews to a given issue, they should be presented to the developer appropriately to support their comprehension of the studied issue. When analyzing an issue related to a bug, the developer should be able to use the information given by the system as a basis to identify for how long the bug has been occurring, as well as more details on the context in which it appears, including causes, steps to reproduce and software and hardware configurations common to the users affected by the problem.

Alternatively, the date and number of pertinent reviews the system matches to an issue describing a new feature should help the developer quantify the persistency and strength of the community's desire for that feature. The system should also help the developers identify possible suggestions and preferences in the content of the pertinent reviews, guiding the choices they make while implementing it.

4.2 Updates to the Implementation

Most of the system architecture was left unchanged from the first implementation cycle. We dedicate this section to discussing the modifications made for the second development phase.

In Figure 4.1 we show an overview of the fundamental steps our revised approach uses to achieve the goal of augmenting the information in issues with reviews.

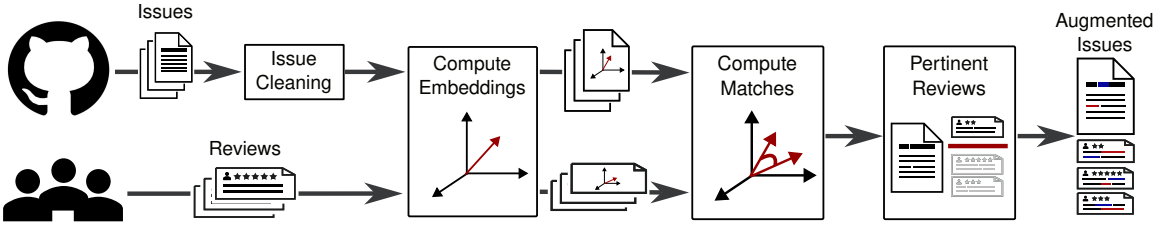

Figure 4.1: *Fundamental components of the revised approach.*

One of the crucial changes made is the inclusion of the issue bodies in the embedding process. While the findings of Haering *et al.*, 2021 showed that the issue titles were efficient at summarizing their contents, it is undeniable that by ignoring the information in the issue bodies, some details on the context are eventually lost. Considering that we chose to use a newer LLM with a larger context size in the second development phase, we decided to include the issue bodies in the embedding.

As discussed after our intermediary evaluation, switching the LLM used for computing the text embeddings could have a meaningful impact on the pertinence of the matches suggested. Besides, a model with a larger context size could also strengthen the approach. We assumed that such a newer model could encode longer text fragments without the introduction of unwanted noise while still producing an embedding that satisfactorily captured all aspects of the issue context.

Therefore, we decided to improve the approach by leveraging the `jina-embeddings-v3` (Sturua *et al.*, 2024) model, a multi-lingual model based on `XLM-RoBERTa` (Conneau *et al.*, 2020), currently ranking as the second best model in the STS task on the MTEB benchmark (Muennighoff *et al.*, 2023). We chose `jina-embeddings-v3` instead of the highest ranked model (`bilingual-embedding-large`[1]) because of its longer context length, capable of embedding issue bodies or even whole discussions with up to 8,194 tokens, and due to its multi-lingual capabilities, not being restricted to English or French, to explore potential benefits for the different languages used by users and developers. Just like the `SentenceTransformers` model used in the first cycle, `jina-embeddings-v3` also produces a single embedding for the whole input text.

Following the results from the first evaluation, it was clear that it might be necessary to filter issues before matching. Accordingly, the new approach allows issues pre-filtering based on specific parameters such as issue labels and tags. We gather their titles and bodies and perform a textual cleaning step. We remove markdown syntax elements, including attachments and typesetting marks (*e.g.*, bold, italics). Besides shortening the content for the embedding context, this step reduces spurious semantic similarities due to recurring patterns in the markdown syntax.

On the other hand, we set aside the review classification and text translation components for the second development cycle. We decided to focus only on English-written issues and reviews to evaluate the approach later without possible interferences caused by the translation processes. Still to reduce the complexity of the approach to focus on the STS-based retrieval, we decided not to filter reviews for the second phase of the project.

From a technical point of view, an essential improvement to the previous implementation was the introduction of checkpoints for persisting embeddings and matches between program executions. As the time taken to compute all embeddings grows significantly with the project popularity (number of reviews) and development history (issues), storing them considerably decreases the approach's computational cost. Furthermore, the cost of computing the similarity between each possible pair of issue and review is even more sensitive to changes in the project size. For a project with m issues and n reviews, the computational cost for measuring the STS for all pairs is $O(mn)$; that is, for a project with 2 million reviews, every new issue in the project reflects on 2 million new comparisons between embeddings. Then, it is only natural that the implementation stores the similarity values computed and matches identified.

However, the most significant change to the approach is the way in which it retrieves matches. As our focus shifted to augmenting issues with pertinent reviews, we inverted the search direction for matching pairs. Instead of looking for issues matching a given review, the system would now retrieve the reviews most similar to a given issue.

4.3 Preparing the Case Study

For the end of the second development cycle, we decided to focus on evaluating the 'quality' of the suggestions brought up by the approach — that is, how frequently they

[1] See **https://tinyurl.com/bilingual-embs**

are relevant and how often the reviews suggested have new information absent on the issue content. To that end, we conduct a case study with the Brave browser Android app [2], a popular FLOSS application with more than 100 million downloads on the Google Play store.[3] We extracted the most pertinent matches our revised approach suggests and conducted a preliminary qualitative analysis to evaluate how useful the suggestions are in augmenting the matched issues.

We collected 9.8k issues and 184k reviews written in English in the United States region. Brave's maintainers use the same GitHub repository for different target operating systems (Android, iOS, Linux, macOS, Windows), so we collected only issues with the *OS:Android* label. After removing the tag `[Android]` from issue titles, we applied the pre-processing steps explained in Section 4.2. We looked for an appropriate value for σ as we analyzed the distribution of issues augmented with at least one match for different similarity thresholds (see Figure 4.2).

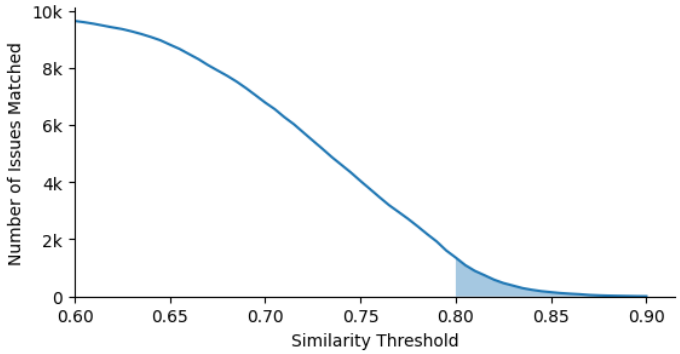

Figure 4.2: *Number of issues augmented for* $\sigma \geq 0.6$.

Starting with a σ similarity threshold of 0.6, our approach would suggest 35 million relevant matches, with at least one match for 9,637 issues (98.2% of all issues). Manual analysis of a subset of suggested reviews shows low pertinence with such a conservative threshold.

We further restricted matches based on the threshold to focus on pertinent matches for at least a handful of issues. After experimentation, we settled with $\sigma = 0.8$. That left us with 7,969 matches that augment 1,359 issues (13.8% of the initial sample) with at least one pertinent match.

To validate whether the reviews with pertinent matches can be useful for augmenting the information present in an issue, we performed a qualitative analysis on the 100 issues with the highest top-1 relevance. Sorting matches by decreasing relevance, we include the first 100 unique issues matched alongside their 5 most relevant matches.

4.4 Case Study Results

Out of the 100 issues analyzed, only 10 were matched to irrelevant or non-pertinent reviews, mostly occurring for unhelpful or shallow issue contents. All the other 90 issues

[2] See **https://github.com/brave/brave-browser**

[3] See **https://play.google.com/store/apps/details?id=com.brave.browser&hl=en-US**

had at least one suggestion pertinent to their contexts, of which 43 had 4 or 5 relevant suggestions.

We present two examples with their top-3 scoring matching reviews to show that they are not only pertinent but also useful as they add meaningful information for the developers.

4.4.1 A Prematurely Closed Bug Report

In Figure 4.3 we show the matches for issue #29130.[4]

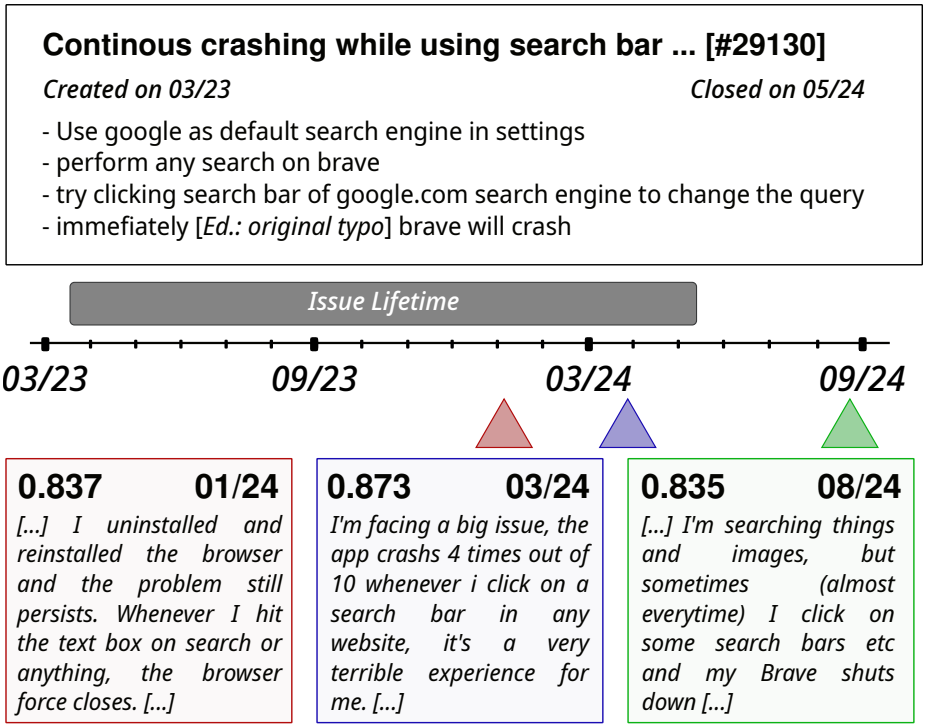

Figure 4.3: *Issue #29130 (Bug Report). Lifetime and Top-3 Relevant Reviews.*

This bug report describes the repeating occurrences of crashes after the users tap a search bar. The top three matches are pertinent to the issue, as they all mention the browser crashing after interacting with search bars. The author of the oldest matching review mentions that the problem also happens for other text input fields, not only search bars. This information is missing from the issue text and discussion, hinting at a potentially latent related bug. Considering that the issue was closed 14 months after it was opened, for being *stale*, the temporal distribution of the matching reviews is interesting.

The first review appears 10 months after the issue was opened, yet the occurrence of crashes for other text input fields is missing in the issue contents and discussion. The second review in order of time, and highest ranking, reports the problem 3 weeks before the issue was closed. The third match found by our approach reports the occurrence of crashes related to search bars 4 months after the issue was closed, also mentioning searching images as causing crashes.

[4] See **https://github.com/brave/brave-browser/issues/29130**

The user tried uninstalling and reinstalling the app, thus implying having the most up-to-date version, letting the developers infer that the bug is still present in that newer version, at least for some hardware, software, and OS configurations.

4.4.2 A Frequently Requested Feature

In Figure 4.4 we see the issue #25863.[5]

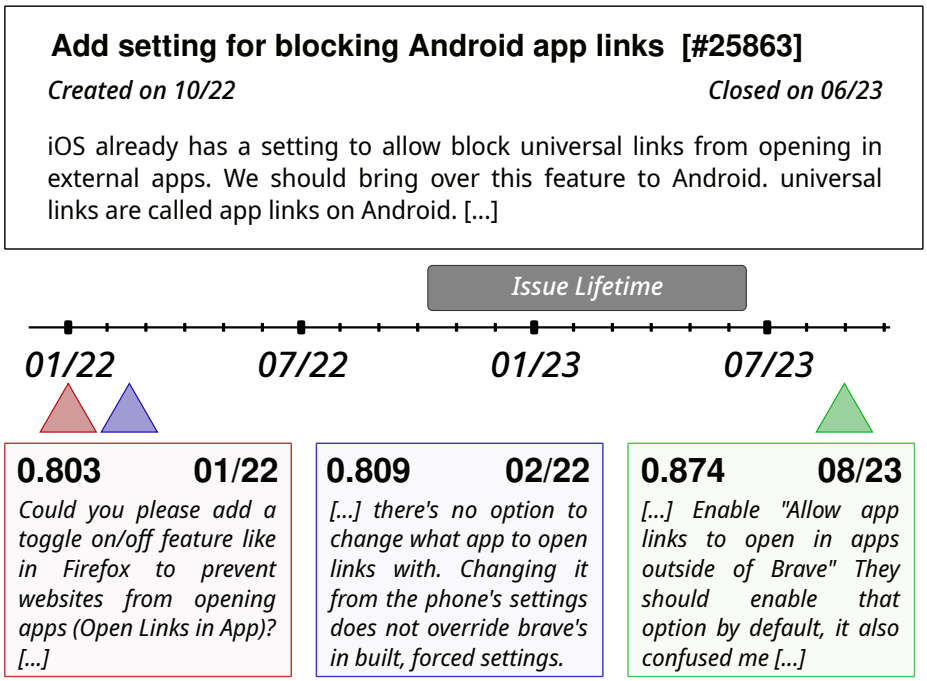

Figure 4.4: *Issue #25863 (Feature Request). Lifetime and Top-3 Relevant Reviews.*

This contains a feature implementation proposal for the option to toggle on or off the behavior of opening links with external apps (actually, a feature already present in Brave for the iOS operating system).

All top-scoring relevant reviews revolve around the expected behavior for opening links in external apps, the core topic of the issue. Each review brings a new insight into how or why the developers should implement the feature. The oldest matching review proves that the feature has been desired for more than 10 months before the issue was opened. The second one brings a concern about the configurability of the default external apps and a misconfiguration, overriding global phone settings. Finally, the most recent review asks for a possible change in the behavior of the feature after the developers have already implemented it, questioning the default value chosen for the feature. The matched reviews come from different moments in time, all outside the lifetime of the issue, each one providing useful information to inform and affect the development process and decisions.

[5] See **https://github.com/brave/brave-browser/issues/25863**

Chapter 5

Discussion

Even though we did not manage to test and evaluate DeeperMatcher alongside developers of big mobile apps, both our preliminary evaluations validate and guide our design choices. Our results indicate the potential that a complete workflow of programmatical STS-based artifact matching retrieval could support crowd-based requirements engineering.

Although the results from the first evaluation cycle were not as quantitatively attractive as those of the second, they are still of great value. Far from suggesting that the approach can not be applied to match issues to a review, the results were essential for providing insights into how we could keep developing and testing the workflow we proposed.

Insight 1
The experimentation we did on the first development cycle elicited improvements to the workflow, some of which could be implemented and tested on the second cycle. The results revealed multiple directions for future improvements, from clustering reviews and issues to prioritizing sections of the artifacts and using multilingual models.

Insight 2
The significant difference between our results and that of the original *DeepMatcher* evaluation suggests that the generalizability of the proof of concept should be tested more thoroughly, considering different projects with different user base profiles and issue-tracking cultures. Only then might one have a solid basis to validate extending the scope of the workflow to other languages and critically distinct data sources.

We can take this discussion further by reflecting on the importance of continuous research in areas as rapidly evolving as LLMs and their applications in software engineering. Our results demonstrate how higher expectations and more advanced models can render recently presented solutions nearly obsolete. Therefore, continuous adaptability should be a guiding principle in designing systems based on LLMs.

For the second evaluation cycle, we expected that a significant part of the issues would be too technical to have pertinent matches with user reviews. While users are more concerned with perceivable behavior and configuration, they are less interested in technicalities. Still, we find that augmenting 14% of the issues with pertinent reviews is an encouraging result that motivates refining and extending our approach, bringing the perspective of real users to the development of popular apps.

The matches found for the 100 issues analyzed stand as undeniable proof that the approach is capable of detecting highly pertinent matches. Moreover, a set of issues augmented by the workflow attests to its potential to enhance issue comprehension. Our qualitative analysis shows that the highest-ranking matches can bring **new insights to support the developers' comprehension of the context of an issue**.

Insight 3
The most pertinent matches to a bug report often include additional information on the context in which the bug manifests itself. The publication date of a bug-related review traces out its lifespan and serves as an early warning or sign of recurring problems.

Insight 4
For new feature requests, the most pertinent matches contain insights on what might better suit the users interests and needs. Time plays a role in helping developers to trace trends or (re-)evaluate past choices based on emerging pertinent and automatically extracted feedback.

Fully automating our approach allows retrospective comparisons of changes in the software product through its development lifecycle and changes in users perception of such changes. We also hypothesize that the multilingual capabilities of the `jina-embeddings-v3` model played a role in overcoming the language differences between the technical writing of developers in issues and how users write their reviews. Nonetheless, our results show that STS and text embeddings can indeed suggest **matching reviews that are useful for augmenting the comprehension of software development issues**.

5.1 Related Work

This work finds itself in the intersection of two different areas of interest: mining and analyzing large amounts of user feedback, and understanding issues, how developers use them and how they affect software development. As such, it is relevant to trace out the state of the art in both fields before moving to the intersection of both.

5.1.1 Analyzing User Feedback

Previous works have focused on studying the prospect of mining app reviews to pinpoint helpful comments or identify trends and recurring topics (Pagano and Maalej, 2013). Some authors focused on classifying reviews or clustering them by topic (Khlifi *et al.*, 2024; Guzman *et al.*, 2015; Panichella *et al.*, 2015; Gao *et al.*, 2018; Chen *et al.*, 2014; Fu *et al.*, 2013; Scalabrino *et al.*, 2019; Stanik *et al.*, 2019), simplifying the identification of trends among large sets of reviews.

In particular, Chen *et al.*, 2014 propose AR-Miner, which not only clusters reviews on the same topic but also filters and ranks reviews by how "informative" they are. Scalabrino *et al.*, 2019 proposed CLAP, a tool that suggests possible emergent user insights through fine-grained review clustering. Panichella *et al.*, 2016 presented AR-Doc, which achieved a precision of 88% for classifying app reviews using feature extraction, structural and sentiment analysis.

5.1.2 Analyzing Development Issues

The broader goal of analyzing and augmenting issue comprehension (Herzig *et al.*, 2013; Fiechter *et al.*, 2021; Grammel *et al.*, 2010) has also received attention. Some authors investigated the characteristics that assist the comprehension and resolution of bug reports (Hooimeijer and Weimer, 2007; Zimmermann *et al.*, 2010), and others studied how artifacts such as issue links (L. Li *et al.*, 2018) and tags (Seiler and Paech, 2017) reflect themselves on how a project is developed. Z. Li *et al.*, 2023 analyzed the impact of issue templates, and Kuramoto *et al.*, 2022 studied whether visual elements (images, videos) influence how long developers take to close an issue.

5.1.3 Comprehending User Feedback Alongside Issues

Some authors have experimented with studying the link between app reviews and issues. Palomba *et al.*, 2015 proposed CRISTAL, an approach that looks for matching issues or commits on reviews already pre-filtered as informative based on traditional information retrieval techniques and the dates of the reviews and issues. However, using dates to filter out relevant issues restricts valuable insights on recurring bugs and requests, which our approach can capture.

Haering *et al.*, 2021 proposed the *DeepMatcher* approach that, although the first to bring the use of LLMs to the task, restricts itself to using only issue titles and pre-filtered bug reports. *DeepMatcher* serves as a tool to identify issues possibly related to a given review describing a problem but also limits itself to only using the embeddings of the nouns found in the review. Thus, the approach we proposed in this work advances the state of the art going one step further.

Instead of exploring app reviews to augment issues, Martens and Maalej, 2019 proposed an approach to automatically create tickets based on context information from tweets containing bug reports. By using an automated chatbot, companies could automatically interact with user tweets and retrieve data such as the user's platform, device, app version, and system version. Then, this data can serve as the basis for newly created entries in

the company's issue tracker.

Also in a similar, yet different context, Tizard *et al.*, 2023 proposed an approach that uses the embeddings generated by the Universal Sentence Encoder (USE) model to perform a semantic search on matches between forum posts and issue tracker entries. Delving further into the study of forum posts, the authors also study the approach while looking for matches with FAQs and other documentation sources.

5.2 Limitations

One of the main limitations of our approach is the fundamental subjectivity of "usefulness". Only stakeholders (*e.g.,* developers, project managers) active in the project can decide if the issues matched to a review belong to the same context, or which reviews suggested effectively help resolve an issue.

Our two single case studies fail to account for variables such as "company" culture and organization of a development team. For example, our approach might not be generalizable to projects using strict issue templates or ones with different stakeholders as authors of issues, as these variables could affect the semantic nature of the issue contents. This also points to a necessary piece of future work: performing an evaluation with actual developers, who are in the best position to judge whether a match is useful or not.

Furthermore, previous work has shown that users from different countries behave differently on their corresponding app stores (S. L. Lim *et al.*, 2015). Hence, a complete evaluation of our workflow should account for this variability and involve apps written by users in various countries. Then, we may validate that the pertinence of the matches suggested overpass possible behavioral and linguistic biases in users from different cultures.

5.3 Future Directions

Firstly, and as previously discussed, the immediate next step after the work we developed is to conduct a more thorough evaluation with the direct involvement of developers and stakeholders responsible for maintaining a mobile application with high enough numbers of issues and user reviews. Besides validating the workflow implemented, future work should consider, implement, and test the points of improvement brought up in the evaluation discussion of our first development cycle.

Moreover, the insights presented by the workflow, extrapolated by collating the separate pieces of information, confirm our understanding that a carefully crafted visual report could magnify the issue augmentation potential. For example, the similarity values and dates involved play a significant role in decoding and utilizing the information presented in the matches suggested.

While using the issue body in addition to its title brings new information to the semantic embedding, in the future, we plan to investigate how the boilerplate used for some types of issues (*e.g.,* bug report templates) could be used to extract the most important sections of each issue body.

To add new information in a recommender fashion, we will evaluate the inclusion of the issue's discussion (*i.e.,* events like linking, commenting, and reacting) into its embedding or include different variables (*e.g.,* review date, software version) into the relevance metric.

As we have chosen a multi-lingual model, future work should also evaluate whether the approach can match reviews in languages different than the one developers use for their issues. Finally, an evaluation with the direct involvement of developers from a popular app is necessary to confirm the generalizability of our approach.

Chapter 6

Conclusion

This capstone project covered the design, implementation, and preliminary evaluation of DeeperMatcher, an innovative system leveraging the growing capabilities of LLMs for crowd-based requirements engineering. The system utilizes semantic text similarity and text embeddings created by LLMs to achieve a previously unfeasible task: matching relevant feedback from large volumes of user data to corresponding issues in the issue tracker.

Its extensible architecture supports various models and the varying preferences of development teams. DeeperMatcher is being developed as a FLOSS command-line utility. It enables developers to identify issues covering the topics of a given user review and augment the comprehension of issues by complementing them with information from automatically identified pertinent reviews.

Having split its development into two distinct phases, we conducted two preliminary evaluations, one using the medium-scale BikeSP project and another using the extremely popular Brave browser app. The results of the first development phase culminated in a publication on the early results track of the 38th Brazilian Symposium on Software Engineering with Prof. Walid Maalej. Similarly, the second development phase counted with the collaboration of Prof. Michele Lanza and resulted in the submission of another short paper to a prestigious international target.

We presented compelling examples of not only pertinent but also useful matches that can inform the development process. Our preliminary results show the potential of the approach and indicate further modifications that might be needed to provide reliable assistance to development teams globally.

Our work underscores the importance of ongoing research into the effective use of LLMs for software and requirements engineering scenarios. Researchers should continuously revisit and reevaluate recent work involving LLMs as newer models and techniques emerge. On the other hand, the Brave browser case study exemplifies how developers seeking to understand an issue context can finally employ a workflow to automatically leverage the wisdom of millions of reviews otherwise left to be forgotten.

References

[Andersen and Maalej 2024] Jakob Smedegaard Andersen and Walid Maalej. "Design patterns for machine learning-based systems with humans in the loop". *IEEE Software* 41.4 (2024), pp. 151–159. doi: 10.1109/MS.2023.3340256 (cit. on p. 14).

[Beck and Andres 2004] Kent Beck and Cynthia Andres. *Extreme programming explained: embrace change*. Addison-Wesley Professional, 2004 (cit. on p. 7).

[Chen *et al.* 2014] Ning Chen, Jialiu Lin, Steven C. H. Hoi, Xiaokui Xiao, and Boshen Zhang. "AR-Miner: Mining informative reviews for developers from mobile app marketplace". In: *Proceedings of the International Conference on Software Engineering (ICSE)*. ACM, 2014, pp. 767–778. doi: 10.1145/2568225.2568263 (cit. on p. 33).

[Cohn 2004] Mike Cohn. *User stories applied: For agile software development*. Addison-Wesley Professional, 2004 (cit. on p. 9).

[Conneau *et al.* 2020] Alexis Conneau *et al. Unsupervised Cross-lingual Representation Learning at Scale*. arXiv:1911.02116. 2020. arXiv: 1911.02116 [cs.CL]. url: https://arxiv.org/abs/1911.02116 (cit. on p. 27).

[Ethayarajh *et al.* 2019] Kawin Ethayarajh, David Duvenaud, and Graeme Hirst. "Towards understanding linear word analogies". In: *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*. Ed. by Anna Korhonen, David Traum, and Lluis Marquez. Florence, Italy: Association for Computational Linguistics, 2019, pp. 3253–3262. doi: 10.18653/v1/P19-1315. url: https://aclanthology.org/P19-1315 (cit. on p. 13).

[Fiechter *et al.* 2021] Aron Fiechter, Roberto Minelli, Csaba Nagy, and Michele Lanza. "Visualizing github issues". In: *Proceedings of the Working Conference on Software Visualization (VISSOFT)*. IEEE, 2021, pp. 155–159. doi: 10.1109/VISSOFT52517.2021.00030 (cit. on p. 33).

[Fu *et al.* 2013] Bin Fu *et al.* "Why people hate your app: making sense of user feedback in a mobile app store". In: *Proceedings of the International Conference on Knowledge Discovery and Data Mining (KDD)*. ACM, 2013, pp. 1276–1284. doi: 10.1145/2487575.2488202 (cit. on p. 33).

[Gao *et al.* 2018] Cuiyun Gao, Jichuan Zeng, Michael R. Lyu, and Irwin King. "Online app review analysis for identifying emerging issues". In: *Proceedings of the International Conference on Software Engineering (ICSE)*. ACM, 2018, pp. 48–58. DOI: 10.1145/3180155.3180218 (cit. on p. 33).

[Grammel *et al.* 2010] Lars Grammel, Holger Schackmann, Adrian Schröter, Christoph Treude, and Margaret-Anne Storey. "Attracting the community's many eyes: An exploration of user involvement in issue tracking". In: *Proceedings of the Workshop on Human Aspects of Software Engineering (HAoSE)*. ACM, 2010. DOI: 10.1145/1938595.1938601 (cit. on p. 33).

[Groen *et al.* 2017] Eduard C. Groen *et al.* "The crowd in requirements engineering: the landscape and challenges". *IEEE Software* 34.2 (2017), pp. 44–52. DOI: 10.1109/MS.2017.33 (cit. on pp. 2, 9).

[Guzman *et al.* 2015] Emitza Guzman, Muhammad El-Haliby, and Bernd Bruegge. "Ensemble methods for app review classification: an approach for software evolution". In: *Proceedings of the International Conference on Automated Software Engineering (ASE)*. IEEE, 2015, pp. 771–776. DOI: 10.1109/ASE.2015.88 (cit. on p. 33).

[Haering *et al.* 2021] Marlo Haering, Christoph Stanik, and Walid Maalej. "Automatically matching bug reports with related app reviews". In: *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. 2021, pp. 970–981. DOI: 10.1109/ICSE43902.2021.00092 (cit. on pp. 2, 13, 15, 17–19, 22, 24–26, 33).

[Herzig *et al.* 2013] Kim Herzig, Sascha Just, and Andreas Zeller. "It's not a bug, it's a feature: how misclassification impacts bug prediction". In: *Proceedings of the International Conference on Software Engineering (ICSE)*. IEEE, 2013, pp. 392–401. DOI: 10.1109/ICSE.2013.6606585 (cit. on pp. 26, 33).

[Honnibal *et al.* 2020] Matthew Honnibal, Ines Montani, Sofie Van Landeghem, Adriane Boyd, *et al. Spacy. Industrial-strength natural language processing in python*. 2020 (cit. on p. 18).

[Hooimeijer and Weimer 2007] Pieter Hooimeijer and Westley Weimer. "Modeling bug report quality". In: *Proceedings of the International Conference on Automated Software Engineering (ASE)*. ACM, 2007, pp. 34–43. DOI: 10.1145/1321631.1321639 (cit. on p. 33).

[IEEE 1990] IEEE. "IEEE Standard Glossary of Software Engineering Terminology". *IEEE Std 610.12-1990* (1990), p. 62. DOI: 10.1109/IEEESTD.1990.101064 (cit. on p. 5).

[Jiao and Zhang 2021] Qilu Jiao and Shunyao Zhang. "A brief survey of word embedding and its recent development". In: *2021 IEEE 5th Advanced Information Technology, Electronic and Automation Control Conference (IAEAC)*. Vol. 5. 2021, pp. 1697–1701. DOI: 10.1109/IAEAC50856.2021.9390956 (cit. on p. 13).

[Julian *et al.* 2019] Brendan Julian, James Noble, and Craig Anslow. "Agile practices in practice: towards a theory of agile adoption and process evolution". In: *Agile Processes in Software Engineering and Extreme Programming*. Springer International Publishing, 2019, pp. 3–18 (cit. on p. 7).

[Khlifi *et al.* 2024] Ghaith Khlifi, Ilyes Jenhani, Montassar Ben Messaoud, and Mohamed Wiem Mkaouer. "Multi-label classification of mobile application user reviews using neural language models". In: *Proceedings of the European Conference on Symbolic and Quantitative Approaches to Reasoning with Uncertainty (ECSQARU)*. Springer, 2024, pp. 417–426. doi: 10.1007/978-3-031-45608-4_31 (cit. on p. 33).

[Kim *et al.* 2017] Yoon Kim, Carl Denton, Luong Hoang, and Alexander M Rush. "Structured attention networks". *arXiv preprint arXiv:1702.00887* (2017) (cit. on p. 11).

[Kuramoto *et al.* 2022] Hiroki Kuramoto *et al.* "Do visual issue reports help developers fix bugs? A preliminary study of using videos and images to report issues on github". In: *Proceedings of the International Conference on Program Comprehension (ICPC)*. ACM, 2022, pp. 511–515. doi: 10.1145/3524610.3527882 (cit. on p. 33).

[L. Li *et al.* 2018] Lisha Li, Zhilei Ren, Xiaochen Li, Weiqin Zou, and He Jiang. "How are issue units linked? empirical study on the linking behavior in github". In: *Proceedings of the Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 2018, pp. 386–395. doi: 10.1109/APSEC.2018.00053 (cit. on p. 33).

[Z. Li *et al.* 2023] Zhixing Li *et al.* "To follow or not to follow: understanding issue/pull-request templates on github". *IEEE Transactions on Software Engineering* 49.4 (2023), pp. 2530–2544. doi: 10.1109/TSE.2022.3224053 (cit. on p. 33).

[S. Lim *et al.* 2021] Sachiko Lim, Aron Henriksson, and Jelena Zdravkovic. "Data-driven requirements elicitation: a systematic literature review". *SN Computer Science* 2 (Feb. 2021). doi: 10.1007/s42979-020-00416-4 (cit. on p. 9).

[S. L. Lim *et al.* 2015] Soo Ling Lim, Peter J. Bentley, Natalie Kanakam, Fuyuki Ishikawa, and Shinichi Honiden. "Investigating country differences in mobile app user behavior and challenges for software engineering". *IEEE Transactions on Software Engineering* 41.1 (2015), pp. 40–64. doi: 10.1109/TSE.2014.2360674 (cit. on p. 34).

[Maalej, Biryuk, *et al.* 2024] Walid Maalej, Volodymyr Biryuk, Jialiang Wei, and Fabian Panse. "On the automated processing of user feedback". In: *Handbook of Natural Language Processing for Requirements Engineering*. Ed. by Alessio Ferrari and Gouri Deshpande. Springer, 2024 (cit. on pp. 9, 17, 24).

[Maalej, Nayebi, *et al.* 2016] Walid Maalej, Maleknaz Nayebi, Timo Johann, and Guenther Ruhe. "Toward data-driven requirements engineering". *IEEE Software* 33.1 (2016), pp. 48–54. doi: 10.1109/MS.2015.153 (cit. on p. 9).

[Martens and Maalej 2019] Daniel Martens and Walid Maalej. "Extracting and analyzing context information in user-support conversations on twitter". In: *International Requirements Engineering Conference (RE)*. IEEE, 2019, pp. 131–141. doi: 10.1109/RE.2019.00024 (cit. on pp. 9, 15, 33).

[Mekala *et al.* 2021] Rohan Reddy Mekala, Asif Irfan, Eduard C. Groen, Adam Porter, and Mikael Lindvall. "Classifying user requirements from online feedback in small dataset environments using deep learning". In: *2021 IEEE 29th International Requirements Engineering Conference (RE)*. 2021, pp. 139–149. doi: 10.1109/RE51729.2021.00020 (cit. on p. 2).

[Mikolov *et al.* 2013] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. "Efficient estimation of word representations in vector space". *arXiv preprint arXiv:1301.3781* (2013) (cit. on p. 13).

[Montgomery *et al.* 2024] Lloyd Montgomery, Clara Lüders, and Walid Maalej. "Mining issue trackers: concepts and techniques". In: *Handbook of Natural Language Processing for Requirements Engineering*. Ed. by Alessio Ferrari and Gouri Deshpande. Springer, 2024. url: https://arxiv.org/abs/2403.05716 (cit. on p. 9).

[Muennighoff *et al.* 2023] Niklas Muennighoff, Nouamane Tazi, Loic Magne, and Nils Reimers. *MTEB: Massive Text Embedding Benchmark*. arXiv:2210.07316. 2023. arXiv: 2210.07316 [cs.CL]. url: https://arxiv.org/abs/2210.07316 (cit. on p. 27).

[Pagano and Maalej 2013] Dennis Pagano and Walid Maalej. "User feedback in the appstore: An empirical study". In: *Proceeding of the International Requirements Engineering Conference (RE)*. IEEE, 2013, pp. 125–134. doi: 10.1109/RE.2013.6636712 (cit. on pp. 2, 9, 33).

[Palomba *et al.* 2015] Fabio Palomba *et al.* "User reviews matter! tracking crowdsourced reviews to support evolution of successful apps". In: *Proceedings of the International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2015, pp. 291–300. doi: 10.1109/ICSM.2015.7332475 (cit. on p. 33).

[Panichella *et al.* 2015] Sebastiano Panichella *et al.* "How can i improve my app? Classifying user reviews for software maintenance and evolution". In: *Proceedings of the International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2015, pp. 281–290. doi: 10.1109/ICSM.2015.7332474 (cit. on p. 33).

[Panichella *et al.* 2016] Sebastiano Panichella *et al.* "Ardoc: app reviews development oriented classifier". In: *International Symposium on Foundations of Software Engineering*. FSE. Seattle, WA, USA: Association for Computing Machinery, 2016, pp. 1023–1027. isbn: 9781450342186. doi: 10.1145/2950290.2983938. url: https://doi.org/10.1145/2950290.2983938 (cit. on p. 33).

[Pilone *et al.* 2024] Arthur Pilone *et al. Projeto Piloto Bike SP*. 2024. url: https://gitlab.com/interscity/bikesp/bikespapp (cit. on p. 19).

[Reimers and Gurevych 2019] Nils Reimers and Iryna Gurevych. "Sentence-bert: sentence embeddings using siamese bert-networks". *arXiv preprint arXiv:1908.10084* (2019) (cit. on p. 18).

[Sammut and Webb 2010] "Tf–idf". In: *Encyclopedia of Machine Learning*. Ed. by Claude Sammut and Geoffrey I. Webb. Boston, MA: Springer US, 2010, pp. 986–987. isbn: 978-0-387-30164-8. doi: 10.1007/978-0-387-30164-8_832. url: https://doi.org/10.1007/978-0-387-30164-8_832 (cit. on p. 10).

[Sanh *et al.* 2019] Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf. "Distilbert, a distilled version of bert: smaller, faster, cheaper and lighter". *arXiv preprint arXiv:1910.01108* (2019) (cit. on p. 17).

[Santos *et al.* 2019] Rubens Ideron dos Santos, Eduard C. Groen, and Karina Villela. "An overview of user feedback classification approaches". In: *REFSQ Workshops*. 2019. url: https://api.semanticscholar.org/CorpusID:186206287 (cit. on p. 2).

[Scalabrino *et al.* 2019] Simone Scalabrino, Gabriele Bavota, Barbara Russo, Massimiliano Di Penta, and Rocco Oliveto. "Listening to the crowd for the release planning of mobile apps". *IEEE Transactions on Software Engineering* 45.1 (2019), pp. 68–86. doi: 10.1109/TSE.2017.2759112 (cit. on p. 33).

[Seiler and Paech 2017] Marcus Seiler and Barbara Paech. "Using tags to support feature management across issue tracking systems and version control systems: a research preview". In: *Proceedings of the International Working Conference on Requirements Engineering: Foundation for Software Quality (REFSQ)*. Springer. Springer, 2017, pp. 174–180 (cit. on p. 33).

[Snijders *et al.* 2014] Remco Snijders, Fabiano Dalpiaz, Mahmood Hosseini, Alimohammad Shahri, and Raian Ali. "Crowd-centric requirements engineering". In: *2014 IEEE/ACM 7th International Conference on Utility and Cloud Computing*. 2014, pp. 614–615. doi: 10.1109/UCC.2014.96 (cit. on pp. 2, 9).

[Stanik *et al.* 2019] Christoph Stanik, Marlo Haering, and Walid Maalej. "Classifying multilingual user feedback using traditional machine learning and deep learning". In: *2019 IEEE 27th International Requirements Engineering Conference Workshops (REW)*. 2019, pp. 220–226. doi: 10.1109/REW.2019.00046 (cit. on pp. 2, 17, 33).

[Sturua *et al.* 2024] Saba Sturua *et al. jina-embeddings-v3: Multilingual Embeddings With Task LoRA*. arXiv:2409.10173. 2024. arXiv: 2409.10173 [cs.CL]. url: https://arxiv.org/abs/2409.10173 (cit. on p. 27).

[Tizard *et al.* 2023] James Tizard, Peter Devine, Hechen Wang, and Kelly Blincoe. "A software requirements ecosystem: linking forum, issue tracker, and faqs for requirements management". *IEEE Transactions on Software Engineering* 49.4 (2023), pp. 2381–2393. doi: 10.1109/TSE.2022.3219458 (cit. on p. 34).

[Valente 2020] Marco Tulio Valente. *Engenharia de Software Moderna*. 2020, p. 395 (cit. on p. 1).

[Vaswani *et al.* 2017] Ashish Vaswani *et al.* "Attention is all you need". *Advances in neural information processing systems* 30 (2017) (cit. on p. 11).

[Wieringa 2014] Roel J. Wieringa. In: *Design Science Methodology for Information Systems and Software Engineering*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014. isbn: 978-3-662-43839-8. doi: 10.1007/978-3-662-43839-8. url: https://doi.org/10.1007/978-3-662-43839-8 (cit. on pp. 2, 20).

[Zimmermann *et al.* 2010] Thomas Zimmermann *et al.* "What makes a good bug report?" *IEEE Transactions on Software Engineering* 36.5 (2010), pp. 618–643. doi: 10.1109/TSE.2010.63 (cit. on pp. 15, 26, 33).

Index