Feature Representations for Semantic Code Retrieval

Merkmalsrepräsentation für semantische Quelltextsuche

Till Wenke

Universitätsbachelorarbeit
zur Erlangung des akademischen Grades

Bachelor of Science
(B. Sc.)

im Studiengang
IT Systems Engineering

eingereicht am 30. September 2024 am
Fachgebiet Artificial Intelligence and Sustainability der
Digital-Engineering-Fakultät
der Universität Potsdam

Gutachter	Prof. Dr. Ralf Herbrich
Betreuer	Prof. Dr. Ralf Herbrich
	Prof. Dr. Gerard de Melo

Abstract

Semantic code retrieval describes the task of finding code based on the content of a natural language query. More specifically, this work is concerned with repository-level code file retrieval from natural language issue descriptions which can be part of broader repository-level code generation pipelines. For this purpose, I apply a Bayesian linear probit regression model to issue-code instances that are represented by interaction features between their two components. The interaction features that I investigate stem from textual representations of the components, the structure of their repository and the past activity in the repository with respect to time and the developers involved. Each of the features was chosen for alone being indicative of a file change. I could show that present text embedding models yield a good semantic representation of natural language as well as code and that a simple linear model can fine-tune a text similarity metric on those embeddings to the specific task of code retrieval. Making use of the interaction features beyond the textual representations did not notably increase or even decreased the model's performance on two single-repository datasets as well as a multi-repository dataset that were specifically compiled for this work.

Zusammenfassung

Semantische Codesuche beinhaltet die Aufgabenstellung, Code auf der Grundlage des Inhalts einer natürlichsprachlichen Anfrage zu finden. Im Speziellen befasst sich diese Arbeit mit dem Auffinden von Codedateien auf Repository-Ebene anhand von natürlichsprachlichen Problembeschreibungen. Zu diesem Zweck wende ich ein Bayes'sches lineares Probit-Regressionsmodell auf Issue-Code-Instanzen an, die durch Interaktionsmerkmale zwischen ihren beiden Bestandteilen dargestellt werden. Die Interaktionsmerkmale, die ich untersuche, stammen aus textueller Repräsentation von Problemstellung und Code, der Struktur ihres Repositorys und der vergangenen Aktivität im Repository in Bezug auf Zeit und die beteiligten Entwickler. Jedes dieser Merkmale wurde ausgewählt, weil es allein auf eine Dateiveränderung hindeutet. Ich konnte zeigen, dass derzeitige Modelle zur Texteinbettung eine gute semantische Repräsentation von natürlichen Sprache und Codes liefern und dass ein einfaches lineares Modell eine Textähnlichkeitsmetrik auf diesen Einbettungen auf die spezifische Aufgabe der Codesuche verfeinern kann. Die Verwendung von Interaktionsmerkmalen, die über die Textrepräsentationen hinausgehen, hat die Leistungsfähigkeit des Modells bei zwei Datensätzen mit einem einzigen Repository und einem Datensatz mit mehreren Repositories, die speziell für diese Arbeit zusammengestellt wurden, nicht nennenswert erhöht oder sogar verringert.

Acknowledgments

I want to thank Jennifer Bonello, Fritz Darchinger, Nico Grimm, Philipp Kolbe, Linus Löll, Jim Maar and Max Rütz for collaborating with me on the Feedback2Code project that lead to this Bachelor's thesis. I want to emphasize the joint work with Nico Grimm on the code retrieval task that also lead to jointly writing Chapters 1 and 2 of this work. I would also like to thank Franziska Dobrigkeit, Dr. Michael Perscheid and Dr. Ralf Teusner for their feedback troughout the work on this project. I am grateful to Prof. Dr. Ralf Herbrich and Prof. Dr. Gerard de Melo for supervision of our project work and this thesis. Lastly, acknowledge support from SAP and Hasso-Plattner-Institut.

Contents

1 Introduction

Automated code generation, which involves non-human actors writing program code, has improved rapidly with advancements in large language models (LLMs) in recent years [Che+21]. Consequently, in-line code completion tools have found wide-spread adoption [Sta23], most famously GitHub's copilot[1]. This suggests that LLMs can as well support developers to convert natural language task descriptions into adequate code changes in large code bases. It is no wonder that this topic recently gained attention in industry solutions like GitHub's copilot workspace[2], Sourcegraph's Cody[3] or the SweepAI[4] tool and academia [Yan+24]. Kalliamvakou [Kal22] shows how such assistants can have a vast impact on developers not only in terms of productivity but also when it comes to satisfaction with their work. Besides that, one should not ignore economic and thus social implications [Elo+23].

In this work, we aim to develop our own natural language task description to code change system to understand how present closed-source solutions might work internally as well as to improve certain sub-systems not least in terms of sustainability.

Environmental impact is a major concern attributed to LLMs caused by the high computing power required [Smi+23]. Present state-of-the-art agentic approaches try to imitate a human programmer and involve LLM calls for every action which results in 8-13x higher monetary costs compared to simpler approaches [Yan+24]. Consequently, we believe that certain components, in particular the process of code retrieval, which involves finding the relevant parts in the code base given a problem, should be attempted using less complex approaches. Several approaches of varying complexity already exist. While retrieval with a simple algorithm such as BM25 is possible in principle [Jim+23], the use of smaller machine learning models upon numerical representations of text (e.g [Nee+22]) is particularly suitable for making more precise statements about the location of the change [SK22]. For the subsequent code change generation task we still leverage large language chat models in an iterative manner.

1 https://github.com/features/copilot
2 https://githubnext.com/projects/copilot-workspace
3 https://sourcegraph.com/cody
4 https://sweep.dev/

To train machine learning models and to evaluate the above described approaches we compile a dataset [Rüt24] that matches natural language technical problem definitions to problem solutions in code on a repository-level. Contributing to the broader research community, we want to create such a dataset containing mainly code written in the Java programming language, in contrast to the SWE-bench benchmark [Jim+23] that only contains Python code. Starting from the code base for the Corona-Warn-App[5] for Germany we collect pairs of task descriptions (called issues) and code changes that solved a task (called pull-requests) from GitHub[6] that make up this dataset.

5 https://github.com/corona-warn-app/cwa-server
6 https://github.com/

2 Motivation: Code Retrieval

In the process of automatically generating a code change from a task description in a software project, eventually it has to be decided which files have to be edited. We find that transformer-architecture-based [Vas+17] large language models (LLMs) are an essential part for fulfilling this task. We define an end-to-end solution as

$$\text{code change} = \text{LLM}(\text{task description}) \tag{2.1}$$

where the code change is the most probable token sequence given the input. We assume that the LLM possesses general world knowledge but not concrete knowledge about the software project at hand, thus the code of the software project has to be made available for the model. In an end-to-end solution, the whole code would be provided to the LLM alongside the task description which poses some serious challenges to it. Firstly, Kamradt [Kam23] suggests that LLMs have difficulties to find relevant information i.e. the code to edit when provided with a large amount of irrelevant text. Secondly, present LLMs can only process a limited amount of text at a time that can be far exceeded by large code bases as for example argued in Jimenez et al. [Jim+23]. Thus, it is vital to reduce the amount of code that qualifies to be edited before handing it to the generative model. For this purpose, we advocate to divide the whole process into two steps in contrast to an end-to-end solution, similar to the common Retrieval Augmented Generation [Lew+20] approach. The two steps comprise finding the files to edit (code retrieval) and editing the files (code generation). Beyond the above-mentioned problems of a one-step approach, regarding the retrieval as a detached step gives us more control over it, such that we can better analyze it and improve its performance.

We can formulate the first step as an information retrieval (IR) task. In general, an IR task consists of a query and a corpus of documents. Each document gets assigned a relevance score which is often binary with respect to the query - the document is either relevant or not relevant for the query. Often, query-document pairs can be thought of as question-answer pairs.

We derive a definition for the code retrieval task from a formalized software engineering process as follows.

Let an issue i be a textual description of a task in a software-engineering process, e.g. fixing a specific bug. This process takes place in a software project that we will as well refer to as a repository in line with common developer jargon. Since software development happens over time, issues are created gradually and can be enumerated. We will thus define any i to be a tuple of a natural number and a string. Secondly, a file f that is part of a software project is usually a structured document containing logical descriptions of abstract objects and/or algorithms that can be interpreted by a computer, which we will, without loss of generality, simply view as a string. We define a repository R_n at state n to be a tuple (I_n, F_n) of a set of unsolved issues I_n and a set of files F_n where the state increases by one every time an issue is added or solved. Every repository starts off as $R_0 = (\emptyset, \emptyset)$. There are principally two operations that can be done on a repository. The first one is to create an issue

$$c : ((I_n, F_n), d) \mapsto (I_{n+1} = I_n \cup ((n+1), d)\}, F_{n+1} = F_n) \tag{2.2}$$

where d is the textual description of a new issue. The second is to solve an issue

$$s : ((I_n, F_n), k, F') \mapsto (I_{n+1} = I_n \setminus I_{n_k}{}^{7}, F_{n+1} = F') \tag{2.3}$$

where k is the identifying number of the solved issue and F' is the updated set of files after solving the issue. With this, we assume that every code change requires an issue that demands it.

To solve the issue identified by k, F' has to be constructed from (I_n, F_n), that involves selecting and editing files from F_n and adding new or deleting existing files. Now, we are only concerned with code retrieval which shall solve the following task:

$$p' : ((I_n, F_n), k) \mapsto F_n^* \subseteq F_n \tag{2.4}$$

that, for any repository at any state R_n given the issue I_{n_k} selects the files that require a change and are handed to a generative model. Our goal then becomes to learn a function

$$p : I_n \times F_n \longrightarrow \{0, 1\} \tag{2.5}$$

that assigns a binary relevance label to every file $f \in F_n$ which should be 1 for every file that needs to be modified in order to solve the issue and 0 otherwise.

In practice, a function $prob : I_n \times F_n \longrightarrow [0, 1]$ is learned using a model M that assigns probabilities to files with respect to being relevant to an issue. When now assigning labels based on a fixed probability threshold (e.g. 50%), we face the problem that a correct F' can only be generated if M is a perfectly accurate model.

7 In this work we denote the selection of an item from a set or list by the item's number as a lower index. What this number is should be clear from the context.

To relax this constraint, we rank files by their predicted probabilities and view the top-k files with the highest probability as relevant, i.e. they are retrieved. Here, k is a natural number and larger than the maximum number of changed files by any single issue. Retrieving the top-k items or all top items that fit in a certain LLM's context window are common approaches that are used in Karpukhin et al. [Kar+20] and Jimenez et al. [Jim+23] respectively. Overall, we are aiming to find a model M that maximizes recall instead of precision. We assume that a generative model that receives multiple files out of which some might indeed not be relevant to resolve an issue, has reasoning capabilities that allow it to make changes that are consistent with each other in the subset of relevant files as we eliminated the Needle in the Haystack problem [Kam23].

We decide for files to be the unit of code that should be retrieved for a few pragmatic reasons. Although there are smaller (e.g. methods and classes) and larger units (e.g. directories) than files in software projects, we find that files are the most natural one from a developer's perspective. Besides that, software development best practices especially for the Java programming language demand that one file corresponds with one class and thus should have a limited size. Problems that may arise with smaller units like methods are that they do not make sense on their own such that we would have to come up with a way to combine them with the necessary environmental information (e.g. imports of the file they are in, signature of the class they belong to). Moreover, we can assume that the number of relevant units relative to their total number decreases the smaller they get which can pose a challenge to machine learning approaches to learn p due to imbalanced data. On the other hand, larger units such as whole directories are not suitable because they can contain many semantically different files that are not really related and their size might easily exceed the context window size of present generative models.

Overall, this work is concerned with investigating feature representations of issues and files that are beneficial to identifying the files that have to be changed to resolve an issue in Java software projects. Beyond classical representations that can be used for code search in general, we are especially focussing on features that can only be obtained from code repositories which distinguish themselves by their structure and history. For this purpose, we as well explor how such features can be leveraged by machine learning models.

3 Methodology: Feature Representations

3.1 Notation

In this work software projects that consistenly stem from GitHub[8] are identified by their organization and repository name as *org-name/repo-name*.

3.2 Preliminary

3.2.1 Function model

In this work I am applying a Bayesian linear probit regression (BLPR) model to predict the probability that a file has to be modified to solve an issue. Those probabilities are used to rank files with respect to being relevant to resolve an issue and to eventually select the relevant files. I chose this model for the following reasons.

- the linear component of the model helps to understand the importance and influence of each feature for the model's predictions - this also makes it a necessity to design features explicitly by hand
- the Bayesian nature of the model allows to specifically model prior knowledge instead of only learning from the data which enables me to operate in a limited data setting e.g. when a model as trained and applied on one repository alone
- uncertainties about the learned weights and the model's predictions can be obtained which can be of value for model development and deployment

As stated in Section 2, the function $prob : I \times F \longrightarrow [0,1]$ should be learned. I expand the definition of issues and files: let an issue $i \in I$ be a tuple of n_i not necessarily numerical features $i = (i_1...i_{n_i})$. Here a feature is a characteristic of an entity. Analogously, a file $f \in F$ is described as $f = (f_1...f_{n_f})$. As the values assigned to a pair (i, f) by $prob$ should be based on the relationship between issue and file, interaction features containing information from the issue and the file at the same

8 https://github.com/

time have to be designed. The reader could imagine telling if a file is relevant for an issue just from their string representations. One would probably compare the strings semantically or look for similar keywords to come to a decision. Exactly this notion of how file and issue should be compared is passed to the non-human model by interaction features. To construct n_c interaction features, for each $j \in \{1...n_c\}$ the function $c_j : (i, f, s_i, s_f) \mapsto o_j(i_{s_i}, f_{s_f})$ with $s_i, s_f \in \mathbb{N}$ selects one feature each from the issue and the file and applies an interaction operation o_j on them that is not further specified here. Throughout this work, I will describe operations o that yield interaction features because the input for o should be clear from the context. As the model that I apply requires a numerical feature vector, each o_j yields a partial feature vector $\mathbf{x}_i$ that makes up the full feature vector $\mathbf{x} = \begin{bmatrix} \mathbf{x}_1 \\ ... \\ \mathbf{x}_{n_c} \end{bmatrix}$. Depending on the specific o_j it could first find a numerical representation for each of its input features and then apply the actual interaction operation or apply the interaction operation as a first step and finish with determining the numerical representation.

$\mathbf{x}$ now represents the issue-file pair that should be consumed by the model. If a full feature vector of l dimensions was constructed, the learning task becomes learning a weight vector $\boldsymbol{w}$ and a bias term b for the linear model:

$$f(\mathbf{x}) = \sum_{i=1}^{l} \boldsymbol{w}_i \cdot \mathbf{x}_i + b \tag{3.1}$$

To eventually yield predictions that can be interpreted as probabilities the values $f(\mathbf{x})$ are mapped to a range from 0 to 1 by the probit link function $\int_0^\infty \mathcal{N}(t; f(\mathbf{x}), \beta^2)dt$ that is applied to it. I set $\beta = \sqrt{\frac{8}{\pi}}$ to approximate a logit link function by matching their derivative at zero.

When applying an error term $e \sim \mathcal{N}(0, \frac{8}{\pi})$ to $f(\mathbf{x})$, this setting can as well be interpreted as calculating the probability of $f(\mathbf{x}) + e$ being larger than 0.

I write $\int_{-\infty}^{x} \mathcal{N}(t; \mu, \sigma^2)dt = \Phi(x; \mu, \sigma^2)$ where the Gaussian cumulative distribution function Φ is parameterized by the mean and variance parameters of the Gaussian - if no parameters are given, the standard Gaussian is assumed for which it holds that $\Phi(x; \mu, \sigma^2) = \Phi(\frac{x-\mu}{\sigma})$ and $\Phi(-t) = 1 - \Phi(t)$.

With this, I can simplify:

$$\int_0^\infty \mathcal{N}(t; f(\mathbf{x}), \beta^2)dt = 1 - \Phi(0; f(\mathbf{x}), \beta^2) \tag{3.2}$$

$$= 1 - \Phi\left(-\frac{f(\mathbf{x})}{\beta}\right) \tag{3.3}$$

$$= \Phi\left(\frac{f(\mathbf{x})}{\beta}\right) \tag{3.4}$$

3.2.2 Setting the Bayesian prior

By working with a Bayesian model, I can set a sensible prior distribution for the model weights which is crucial for the data-scarce task at hand and helps to encode our prior domain knowledge. Prior weight distributions can be thought of as one's belief about the model weights before seeing the data for a specific task. I aim to set an informative prior weight distribution for each dimension of the feature vector space that the model is trained on. Here I refer to an informative prior as the opposite of a weakly informative prior, that if a Gaussian prior is used is distributed according to $\mathcal{N}(0, 10^n)$ with $n \in \mathbb{N}$ and neglects any prior knowledge about the distribution of weights. Most of the features that I will design are either difficult to interpret or I do not have access to prior knowledge about how they relate to the target variable other than a rough trend. Because of that, I will make use of some dataset specific statistics to set the prior weight distributions. This is not considered a perfect Bayesian workflow which I justify in Section 3.2.2.

For a specific dataset D consisting of samples $\mathbf{X}$ and targets $\mathbf{y}$ the portion of samples with positive target variables is known as $\mathrm{P}(y = 1) = p$. Now, the prior weights for the model should be set such that for the single samples in $\mathbf{X}$, p is predicted which is the best naive prediction for a model that did not learn from the data yet.

Assuming the prior weights are $\boldsymbol{w} \sim \mathcal{N}(\boldsymbol{\mu_w}, \Sigma_{\boldsymbol{w}})$ and there is a bias term which is $b \sim \mathcal{N}(\mu_b, {\sigma_b}^2)$, predictions on a single sample $\mathbf{x}$ can be formalized as follows:

$$\mathrm{P}(y = 1|\mathbf{x}) = \int_0^\infty \left[\int \mathcal{N}(t; f(\mathbf{x}, \boldsymbol{w}), \beta^2) \cdot \mathcal{N}(\boldsymbol{w}; \boldsymbol{\mu_w}, \Sigma_{\boldsymbol{w}}) d\boldsymbol{w}\right] dt \tag{3.5}$$

with ${\boldsymbol{\sigma_w}}^2$ being the diagonal of $\Sigma_{\boldsymbol{w}}$ and marginalizing over $\boldsymbol{w}$ this gives

$$\mathrm{P}(y = 1|\mathbf{x}) = \int_0^\infty \mathcal{N}(t; \sum_{i=1}^{n} \mathbf{x}_i \cdot \boldsymbol{\mu_w}_i + \mu_b, \sum_{i=1}^{n} \mathbf{x}_i^2 \cdot {\boldsymbol{\sigma_w}}_i^2 + {\sigma_b}^2 + \beta^2) dt \tag{3.6}$$

$$= \Phi\left(\frac{\sum_{i=1}^{n} \mathbf{x}_i \cdot \boldsymbol{\mu_w}_i + \mu_b}{\sum_{i=1}^{n} \mathbf{x}_i^2 \cdot {\boldsymbol{\sigma_w}}_i^2 + {\sigma_b}^2 + \beta^2}\right) \tag{3.7}$$

Now, $\boldsymbol{\mu_w}$, $\boldsymbol{\sigma_w}$, μ_b and σ_b have to be set such that:

$$p = \Phi\left(\frac{\sum_{i=1}^{n} \mathbf{x}_i \cdot \boldsymbol{\mu_w}_i + \mu_b}{\sum_{i=1}^{n} \mathbf{x}_i^2 \cdot \boldsymbol{\sigma_w}_i^2 + {\sigma_b}^2 + \beta^2}\right) \tag{3.8}$$

As only one of the parameters can be exactly determined from this, I fix the remaining three. I choose to exactly determine μ_b because it is responsible for calibrating the model predictions for a sample that does not contain any information (i.e. $\mathbf{x} = \mathbf{0}$). $\boldsymbol{\mu_w}$, $\boldsymbol{\sigma_w}$ are fixed with regard to knowledge about the feature vector dimensions of $\mathbf{x}$ and $\sigma_b = 1$ as a default choice.

As a consequence, μ_b can be determined by rearranging as follows:

$$\Phi^{-1}(p) = \frac{\sum_{i=1}^{n} \mathbf{x}_i \cdot \boldsymbol{\mu_w}_i + \mu_b}{\sum_{i=1}^{n} \mathbf{x}_i^2 \cdot \boldsymbol{\sigma_w}_i^2 + {\sigma_b}^2 + \beta^2} \tag{3.9}$$

$$\Phi^{-1}(p) \cdot \sum_{i=1}^{n} \mathbf{x}_i^2 \cdot \boldsymbol{\sigma_w}_i^2 + {\sigma_b}^2 + \beta^2 = \sum_{i=1}^{n} \mathbf{x}_i \cdot \boldsymbol{\mu_w}_i + \mu_b \tag{3.10}$$

$$\mu_b = \Phi^{-1}(p) \cdot \sum_{i=1}^{n} \mathbf{x}_i^2 \cdot \boldsymbol{\sigma_w}_i^2 + {\sigma_b}^2 + \beta^2 - \sum_{i=1}^{n} \mathbf{x}_i \cdot \boldsymbol{\mu_w}_i \tag{3.11}$$

Calculating μ_b for all samples in $\mathbf{X}$, I get a dataset of μ_b from which I choose the sample mean as the μ_b of the prior.

I expand the above scheme by one more feature vector dimension $n+1$ for which I assume that it is more meaningful for predicting the target variable than the other feature vector dimensions. I let $\boldsymbol{w}_{n+1} = w \sim \mathrm{N}(\mu, \sigma^2)$.

With this, I get:

$$\mathrm{P}(y = 1|\mathbf{x}) = \Phi\left(\frac{\sum_{i=1}^{n} \mathbf{x}_i \cdot \boldsymbol{\mu_w}_i + \mathbf{x}_{n+1} \cdot \mu + \mu_b}{\sum_{i=1}^{n} \mathbf{x}_i^2 \cdot \boldsymbol{\sigma_w}_i^2 + \mathbf{x}_{n+1}^2 \cdot \sigma^2 + {\sigma_b}^2 + \beta^2}\right) \tag{3.12}$$

where μ and μ_b are the parameters that should be determined, σ is fixed at 1 and the other parameters are fixed as above.

For a meaningful $\mathbf{x}_{n+1}$ I assume without loss of generality that

$$\mathrm{P}(y = 1|\mathbf{x}_{n+1} \leq a_1) < \mathrm{P}(y = 1) < \mathrm{P}(y = 1|\mathbf{x}_{n+1} \geq a_2) \tag{3.13}$$

with $a_1, a_2 \in \mathbb{R}, a_1 < a_2$. This can be seen as the value of $\mathbf{x}_{n+1}$ introducing another bias to the model.

For a binary $\mathbf{x}_{n+1}$ this becomes

$$P(y = 1|\mathbf{x}_{n+1} = 0) < P(y = 1) < P(y = 1|\mathbf{x}_{n+1} = 1) \tag{3.14}$$

for which I define $p_0 = P(y = 1|\mathbf{x}_{n+1} = 0)$ and $p_1 = P(y = 1|\mathbf{x}_{n+1} = 1)$.

Consequently, for samples where $\mathbf{x}_{n+1} = 0$ I can determine μ_b by:

$$p_0 = \Phi\left(\frac{\sum_{i=1}^{n} \mathbf{x}_i \cdot \boldsymbol{\mu_w}_i + \mu_b}{\sum_{i=1}^{n} \mathbf{x}_i^2 \cdot \boldsymbol{\sigma_w}_i^2 + \sigma_b{}^2 + \beta^2}\right) \tag{3.15}$$

and $\mu + \mu_b$ from samples where $\mathbf{x}_{n+1} = 1$ by:

$$p_1 = \Phi\left(\frac{\sum_{i=1}^{n} \mathbf{x}_i \cdot \boldsymbol{\mu_w}_i + (\mu + \mu_b)}{\sum_{i=1}^{n} \mathbf{x}_i^2 \cdot \boldsymbol{\sigma_w}_i^2 + \sigma^2 + \sigma_b{}^2 + \beta^2}\right). \tag{3.16}$$

For all samples where $\mathbf{x}_{n+1} = 0$ one can say that $\mu_b \sim d_0$ and $\mu + \mu_b \sim d_1$ from the samples where $\mathbf{x}_{n+1} = 1$. I am setting the prior parameters for the model according to the means of the distributions. Therefore, the prior parameters can trivially be obtained from $\mu_b = \mu_{d_0}$ and $\mu + \mu_b = \mu_{d_1}$.

For continuous $\mathbf{x}_{n+1}$ I follow an analogous scheme where $p_0 = P(y = 1|\mathbf{x}_{n+1} \le a_1)$ and $p_1 = P(y = 1|\mathbf{x}_{n+1} \ge a_2)$ and a_1, a_2 are chosen to be the first and the last q-quantile with respect to the values $\mathbf{X}_{n+1}$. I denote the dataset of values $\mathbf{x}_{n+1} \le a_1$ as $\mathbf{X}_{n+1}^{a_1}$. For samples where $\mathbf{x}_{n+1} \le a_1$, I approximate $\mathbf{x}_{n+1}$ as $\overline{\mathbf{X}_{n+1}^{a_1}}$. With

$$p_0 = \Phi\left(\frac{\sum_{i=1}^{n} \mathbf{x}_i \cdot \boldsymbol{\mu_w}_i + \overline{\mathbf{X}_{n+1}^{a_1}} \cdot \mu + \mu_b}{\sum_{i=1}^{n} \mathbf{x}_i^2 \cdot \boldsymbol{\sigma_w}_i^2 + \sigma^2 + \sigma_b{}^2 + \beta^2}\right). \tag{3.17}$$

this yields a distribution of $\overline{\mathbf{X}_{n+1}^{a_1}} \cdot \mu + \mu_b \sim d_0$. As above, it should hold that $\overline{\mathbf{X}_{n+1}^{a_1}} \cdot \mu + \mu_b = \mu_{d_0}$. Equivalently, the samples where $\mathbf{x}_{n+1} \ge a_2$ yield $\overline{\mathbf{X}_{n+1}^{a_2}} \cdot \mu + \mu_b = \mu_{d_1}$. By solving this linear system of equations, one can calculate the prior parameters

$$\mu = \frac{\mu_{d_0} - \mu_{d_1}}{\overline{\mathbf{X}_{n+1}^{a_1}} - \overline{\mathbf{X}_{n+1}^{a_2}}} \tag{3.18}$$

$$\mu_b = \mu_{d_1} - \overline{\mathbf{X}_{n+1}^{a_2}} \cdot \frac{\mu_{d_0} - \mu_{d_1}}{\overline{\mathbf{X}_{n+1}^{a_1}} - \overline{\mathbf{X}_{n+1}^{a_2}}} \tag{3.19}$$

Setting the Bayesian prior with respect to a dataset

To set the prior distribution for the model weights as described above, I make use of dataset-specific statistics. This is not considered a perfect Bayesian approach as the prior should only contain information one has access to before seeing the specific data such as common knowledge or domain expertise. Here, I am presenting an information-theoretic argument why this still does not violate the Bayesian approach.

Let us establish a setting in which the performance of a model in a binary classification task on the test set is determined by the accuracy of predicting the correct class label. The performance metric is measured with a precision of n floating point digits such that the real value of the metric can vary by $0.1^n - \epsilon \approx 0.1^n$. Now, if one knows the label of a portion of 0.1^n of the test samples in advance and uses it instead of a model's predictions, it is possible that the value of the accuracy metric does not change (even if the model did not correctly predict those samples before).

I assume that knowing the label of a single sample from a dataset requires its index and 1 bit for the binary label. This can be considered an efficient way to store labels for a small portion of a dataset. For a dataset of s samples, encoding the index requires $\log_2 s$ bits. Thus, in the above scenario, $0.1^n \cdot s \cdot (\log_2 s + 1)$ bits of information were extracted without affecting the performance metric. I call this the bit threshold t. It is implied that equally extracting t' bits, where $t' \leq t$, from the training dataset before training will not improve the model to a measurable extent. Consequently, I allow extracting this amount of bits to inform the prior setting without violating the Bayesian workflow.

Instead of actual labels per sample, one can also extract statistics about the training dataset that are floating point numbers altogether represented by t' bits. If, to set the prior with respect to a dataset and n_v statistical values are measured, each of those values is allowed to be represented as a maximum $\lfloor \frac{t}{n_v} \rfloor$ bit floating point number.

3.2.3 Evaluation

The code retrieval task is concerned with issues that are possibly dependent on each other for one repository which sets up some constraints for how this task can be evaluated. Besides that, I will introduce some performance metrics in this section that are specific to information retrieval tasks.

The code retrieval task was defined as $p' : ((I_n, F_n), k) \mapsto F_n^* \subseteq F_n$ thus selecting the files that have to be changed given an issue and the current state of files in a repository. Imagine a setting in which a not further defined model was trained on a state of the repository and the test dataset contained another earlier state of the repository. Just by comparing the two states, the model could determine all files that are similar and exclude them from the set of possibly relevant files, as they likely never changed between the two states. This is just one example, showcasing that evaluation has to be performed such that it ensures that samples seen during training occurred before validation samples and validation samples before test samples to prevent data leakage. Or intuitively speaking, it does not make sense to learn from the future of a repository when testing on the past, as the training data contains traces of the past.

I find that the following evaluation metrics are relevant to measure model performance in the code retrieval task.

As outlined in Section 2, for an issue instance I_{n_k}, the files F_n are ranked by the predicted probability of being relevant, with the highest probability at the first position. Each of the files has a ground truth binary relevance label and rel_i is the relevance label for the file that is ranked at the i-th position by the predictive model. $n = |F_n|$ is the number of files in the repository at the time of solving a specific issue. From this, one can define the discounted cumulative gain

$$\text{DCG} = \sum_{i=1}^{n} \frac{rel_i}{log_2(i+1)}, \tag{3.20}$$

ideal discounted cumulative gain where $irel_i$ is defined similar as rel_i but samples are ranked according to their ground truth relevance labels instead of the model predictions

$$\text{iDCG} = \sum_{i=1}^{n} \frac{irel_i}{log_2(i+1)} \tag{3.21}$$

and normalized discounted cumulative gain

$$\text{nDCG} = \frac{\text{DCG}}{\text{iDCG}}. \tag{3.22}$$

With the number of issues t, I report AveNDCG

$$\frac{\sum_{i=1}^{t} \text{nDCG}}{t} \tag{3.23}$$

as a general goodness measure of the ranking which is independent of how many files I decide to retrieve.

As stated in 2, a positive relevance label is assigned to the top-k files independent of their predicted probability which then can be compared to the ground truth labels.

I report average recall (AveR@k)

$$\frac{\sum_{i=1}^{t} \text{Recall@k}}{t} \tag{3.24}$$

with

$$\text{Recall@k} = \frac{\#\text{true positive}}{\#\text{true positives} + \#\text{false negative}} \tag{3.25}$$

which should be the metric that is of most interest as it correlates to how well the issue can be resolved by the generative step assuming that each file is equally important for the issue. Moreover, it is easier to interpret in comparison to AveNDCG.

In contrast, I report average precision (AveP@k)

$$\frac{\sum_{i=1}^{t} \text{Precision@k}}{t} \tag{3.26}$$

with

$$\text{Precision@k} = \frac{\#\text{true positive}}{\#\text{true positives} + \#\text{false positives}} \tag{3.27}$$

which is of less importance as described in 2.

Eventually, I want to know if all relevant files were retrieved for an issue and report average "all found" (AAF) as

$$\frac{\sum_{i=1}^{t} \mathbb{1}(\text{Recall@k} = 1.0)}{t}. \tag{3.28}$$

This metric is relevant for the subsequent code generation step as it states an upper bound for the portion of issues that can be fully resolved.

3.3 Text-based features

In the following, I describe the interaction features that I design to represent issue-file pairs.

First of all, text-based features should represent what a human can do when given just a file in their preferred file viewer and an issue in the issue tracking system that is used for that software project. The human is tasked to decide whether the file has to be changed to resolve the issue. In this setting, a file f is represented by a tuple (f_p, f_c) of the path to the file in the software project and its string content.

The file path most importantly contains the file name and type. On the other hand, the issue i is represented by the tuple (i_t, i_d) which is the title of the issue and its description. Here the title is a short summary of the description that should contain the most important information from the description as seen in popular software project task tracking solutions like GitHub[9] or GitLab Issues[10]. The description is a more detailed, often structured text that describes a task related to the software project. I am reasoning that one might not need the entire file content to estimate its relevance with respect to an issue. Instead, just part of the file's content that is extracted via the filtering function $g : f_c \mapsto f_c'$ could be sufficient. Explicitly, I propose to investigate representing a file by the comments contained in it because they are used to summarize and explain code which is often easier to understand (for humans) than the code itself.

As information retrieval using only text-based features is a common task that has been researched and applied in commercial products a lot, I will first present two approaches that make use of text-based features but not of the BLPR model that should serve as baselines.

3.3.1 Bag-of-words-based representation

As a first and also baseline approach that only leverages text-based features I am applying the BM25 [RZ09] ranking function. It is a common non-learning approach to information retrieval and as such is often used in practice in search engines and in academia as a baseline for recent neural-network-based retrieval methods [Jim+23; Kar+20]. BM25 is a probabilistic model that given a query assigns a probability to each document from a corpus of documents for being relevant with respect to the query. This method is based on word counts within a single document and across the whole corpus but not the relationships between words. Thus, it might fail to capture broader semantics.

As the BM25 system was designed with natural language queries and documents in mind [Rob+94] some doubts can arise that it will be able to handle documents consisting of structured text e.g. code well because keywords in the two types of text might be different.

9 https://github.com/features/issues
10 https://docs.gitlab.com/ee/user/project/issues/

3.3.2 Semantic representation

Bag-of-words based methods can fail to detect the semantics of text for example in the case of double negation. Thus, more advanced methods have to be applied to capture the meaning of text.

For this purpose I turn towards text embedding models that represent text as vectors in high dimensional space. These vectors can be designed such that the distance between them is a measure for semantic similarity of the texts they represent. Thus, a vector alone also represents the semantic of its corresponding text. This makes text embeddings a well suited method in information retrieval systems.

Background

I want to lay some foundation about how text embeddings are obtained to improve the understanding for how they can be used in the context of code retrieval and as features in Bayesian machine learning models.

Recent text embedding models stem from pre-trained transformer-architecture-based [Vas+17] large language models [Lee+24]. There are several approaches to obtain a vector representation from such LLMs that can serve as a semantic representation of the text that is put into it. These LLMs consider the input text as a sequence of tokens, multiple characters that words consist of. Neelakantan et al. [Nee+22] use the last token embedding as the text embedding whereas Lee et al. [Lee+24] propose to get the contextualized per token embeddings [Ala18] from one of the LLM layers and aggregate them into one embedding for example via mean pooling.

Now, the LLM is fine-tuned or trained from scratch to yield embedding representations given a dataset of ground truth text pairs that are known to be similar in meaning. For this purpose, a contrastive loss function is applied that for each batch of training samples favors that the learned embeddings of semantically similar texts are closer together than for texts that are semantically unrelated according to the ground truth data. In many popular works [Gün+24; Iza+22; Lee+24] this is achieved by minimizing the InfoNCE [OLV19] loss function. For a pair (a, b) of semantically similar texts and a list $(n_1 ... n_N)$ of negative texts that are not similar to a it can be formulated as:

$$-\frac{e^{sim(a,b)}}{e^{sim(a,b)} + \sum_{i=1}^{N} e^{sim(a,n_i)}} \tag{3.29}$$

where $sim(a, b)$ is the semantic similarity measure of two texts a and b that is defined as the cosine similarity of the vector representations $\mathbf{a}$ and $\mathbf{b}$ of the texts.

Cosine similarity as the cosine of the angle between two vectors is defined as:

$$sim(\mathbf{a}, \mathbf{b}) = \frac{\mathbf{a} \cdot \mathbf{b}}{||\mathbf{a}|| \cdot ||\mathbf{b}||} \tag{3.30}$$

As the length of the vectors is not relevant to cosine similarity as a distance measure between two vectors, popular text embedding models produce unit length embedding vectors. Thus, I can use this property of embedding vectors when working with them and imagine them as points on a hypersphere.

For unit length vector the similarity metric becomes:

$$sim(\mathbf{a}, \mathbf{b}) = \mathbf{a} \cdot \mathbf{b} \tag{3.31}$$

For the code retrieval task in this work it is crucial that the learned embeddings can not only be used to measure the semantic similarity between two natural language texts such that similar texts could be used interchangeably. Instead, the relationship between issue and a code file can rather be characterized as question-answer or problem-solution. These relationships should also be considered as semantic similarity. In addition, the embedding model should have the ability to equally well embed natural language and code.

There are two main properties of an embedding model that can be examined to analyze whether the above criteria are fulfilled.

- Empirical performance of the model on:
  - information retrieval benchmarks such as Beir [Tha+21], especially question-answer retrieval tasks
  - code-search-from-natural-language benchmarks such as Code Search Net [Hus+20]
  - specific code information retrieval benchmarks like CoIR [Li+24]
- Inherent assumed abilities according to the training dataset. I assume that all major general purpose embedding models use similar or the same approaches to collect pairs of natural language and code that are considered similar in semantic like the following:
  - the "top-level docstring in a function along with its implementation as a (text, code) pair" as used by Neelakantan et al. [Nee+22]
  - coding questions with their answers from platforms such as Stack Overflow[11] as used by Lee et al. [Lee+24] and Ni et al. [Ni+22]

11 https://stackoverflow.com/

Ultimately, embeddings obtained from a model that follows the above criteria can confidently be applied to measure the relevance of a file with respect to an issue by cosine similarity. Consequently, this approach can be used in code retrieval applications without further fine-tuning the embeddings to a specific task.

Embeddings in the BLPR model

The just described cosine similarity on semantic embedding vectors can be seen as a linear model.

$$sim(\mathbf{a}, \mathbf{b}) = \mathbf{a} \cdot \mathbf{b} = \sum_{i=1}^{l} \mathbf{a}_i \cdot \mathbf{b}_i = \sum_{i=1}^{l} \mathbf{a}_i \cdot \mathbf{b}_i \cdot \boldsymbol{w}_i + b = \sum_{i=1}^{l} \mathbf{x}_i \cdot \boldsymbol{w}_i + b \tag{3.32}$$

where embedding vectors have a length l, $\boldsymbol{w}$ is default weight vector of ones, b is the bias term which is 0 by default and $\mathbf{x} = \mathbf{a} \odot \mathbf{b}$ with the Hadamard product $\odot$ becomes the feature vector.

According to the embedding model that was used to generate the embedding vector, applying these default weights should result in a good performance on a wide range of semantic similarity tasks. The fact that I only want to make use of the embeddings for one of the tasks they can be used for, lead me to the following insight.

I naively assume that the dimensions of embedding vectors represent interpretable features with regard to the text they represent. Of those interpretable features, some are more relevant to our task than others. Here, by interpretable I mean that the feature has a meaning on its own, in contrast to multiple features together conveying a meaning. Now, interpreting each dimension as an independent feature, a weight vector $\boldsymbol{w}$ can be learned that is more specific to the task in this work.

Based on this, the operation o that generates the semantic interaction feature between issue i and file f is defined as:

1. applying the embedding model M to get $\mathbf{e}_i = M(i)$ and $\mathbf{e}_f = M(f)$ as numerical representations of the respective text
2. establish the interaction between the semantic representations by constructing the feature vector $\mathbf{x} = \mathbf{e}_i \odot \mathbf{e}_f$

I will refer to the feature that is represented by this feature vector $\mathbf{x}$ as the *semantic similarity feature.*

I am providing an illustration in Figure 3.1, to improve the understanding of what learning $\boldsymbol{w}$ means with respect to transforming the embedding vector space. Neglecting the bias term in the linear model and assuming $\forall i \in \{1...l\} : \boldsymbol{w}_i \geq 0$ the linear model can be written as follows:

$$\sum_{i=1}^{l} \mathbf{x}_i \cdot \boldsymbol{w}_i = \sum_{i=1}^{l} \mathbf{a}_i \cdot \sqrt{\boldsymbol{w}_i} \cdot \mathbf{b}_i \cdot \sqrt{\boldsymbol{w}_i} \tag{3.33}$$

Which can be seen as scaling the i-th dimension of the embedding space by $\sqrt{\boldsymbol{w}_i}$ as shown in Figure 3.1. This transformation of a vector $\mathbf{v}$ from the embedding space can be achieved by

$$t(\mathbf{v}) = \begin{bmatrix} \sqrt{\boldsymbol{w}_1} & . & 0 \\ . & . & . \\ 0 & . & \sqrt{\boldsymbol{w}_l} \end{bmatrix} \cdot \mathbf{v} \tag{3.34}$$

This shows that my approach only allows simple transformations of the embedding space and that more complex transformations could for example be achieved by introducing a feature $\mathbf{x}_{l+1} = \mathbf{a}_i \cdot \mathbf{b}_j$ where $i \neq j$. Note that for a $\boldsymbol{w}_i < 0$ the respective addend would become $\mathbf{a}_i \cdot (-\sqrt{\boldsymbol{w}_i}) \cdot \mathbf{b}_i \cdot \sqrt{\boldsymbol{w}_i}$ which can be understood as flipping one axis for one of the vectors as seen in Figure 3.1.

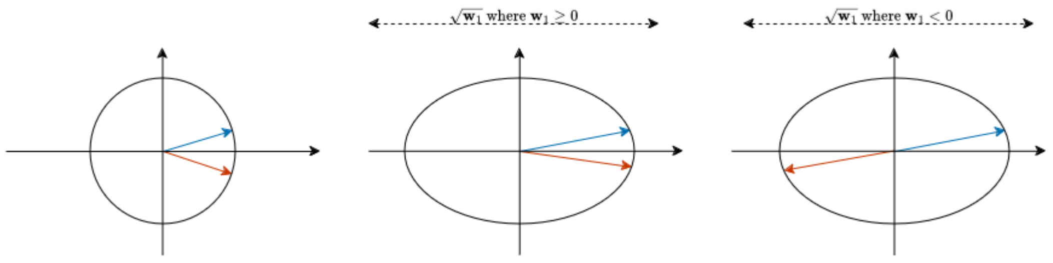

$\sqrt{\mathbf{w}_1}$ where $\mathbf{w}_1 \geq 0$
$\sqrt{\mathbf{w}_1}$ where $\mathbf{w}_1 < 0$

Figure 3.1: Illustration of transformations of a simple embedding vector space on a 1-sphere (circle) achieved by the linear model. Default setting (left) and learned weights (middle, right).

3.3.3 Software-project-structure-based features

There might be several relationships between issue and file that are not captured by the semantic similarity of the text alone. For this purpose, interaction features have to be designed in a more explicit fashion. In this section, I am focusing on features that use the issue text on the one hand and the structure of the repository in the form of the path to a file on the other hand. This follows the notion that for a human developer, it might be enough to look at directories and file names to determine whether a file contains parts that should be modified to resolve an issue.

File-directory-based features

Given an issue, the most obvious thing for a human programmer to identify might be in which sub system of the software project a change has to be made. Thus, identifying the right subsystem can reduce the search space of files that are examined in more detail a lot. One level of granularity of sub systems can be the top level directory of a file. This is what I am concerned with here. In Appendix A I present an experiment that indicates that issues contain information with respect to the top level directory of files.

A top level directory is one that can be found at the root of a repository including the root itself which is the top level directory for all files that are not part of any directory. These directories are rarely the same across repositories e.g. amongst java repositories in Rütz [Rüt24] or *corona-warn-app/cwa-server* some but not all have a src/, script(s)/ or doc(umentation)/ directory. As a consequence, this interaction feature can only be constructed and used for a single repository.

I define the operation o that generates the *top level directory (interaction) feature* between issue i and file f as follows. Let T be the set of top level directories in one specific repository.

1. applying the embedding model M to get $\mathbf{e}_i = M(i)$ as a numerical representation of the issue and let t be the top level directory with regards to f
2. to represent the categorical feature t numerically, it is one-hot encoded by a feature vector $\mathbf{t}$ of length $|T|$
3. establish the interaction between the issue description and the top level directory of the file by constructing the feature vector $\mathbf{x} = \begin{bmatrix} \mathbf{x}_1 \\ \dots \\ \mathbf{x}_{|T|} \end{bmatrix}$ where $\mathbf{x}_k = \mathbf{e}_i \cdot t_k$ with $k \in \{1 \dots |T|\}$ – to rephrase $\mathbf{x}_k$ holds the embedding vector of the issue if t is the k-th top level directory and otherwise zeros

For embedding vectors with l dimensions and repositories with $|T|$ top level directories this results in a $l \cdot |T|$ dimensional feature vector for this interaction feature alone. Especially in the limited data setting of this work this can be a too large feature vector space and one can easily fall to the curse of dimensionality.

To cope with this problem, I only select the n_t most important top level directories and consider files that are not in one of those directories as part of a $(n_t + 1)$-th directory. I define a directory to be important if files contained in it get changed a lot which implies that most of the developer's attention is put on this directory and that it shapes the software project significantly. Most software projects get initialized with a first rough structure and a few files like the repositories from Rütz [Rüt24]. Thus, I assume that with access to a domain expert, the n_t most important top level directories can be regarded as given a priori without looking into a specific dataset.

File-path-based features

Paths to files or other identifiers that lead to a file can occur in an issue for various purposes. They might be part of a stack trace that is related to an error that is the reason for the issue, the author of the issue includes it specifically to point out where a problem lies or where the first place to start with a new feature would be, just to name a few. Some of those mentions are directly related to one of the files that should be changed to resolve the issue, others might be misleading e.g. when files are given as an example or if it is a reference to a method that should be called.

Table 3.1 lists a few possibilities that I can imagine how a file is referenced in an issue other than just its full file path.

Based on this, the operation o can generate a binary *file path feature*:

1. let i be the description of an issue and p the path to a file f
2. extract a variant p' from p according to Table 3.1
3. the feature is active (1) if p' is a substring of i otherwise it is inactive (0)

From those features, I find it is especially worth to deeper investigate those that are highly indicative of a file change and frequently present which is examined in Section 4.2.4.

3.4 Software-engineering-process-based features

So far, the features that I designed viewed issues as rather independent items, not as a step in a software engineering life cycle. The features presented in this section should embed the issues and files in question into the environment of their software project that I understand as software engineers (people) collaborating towards a goal over time.

Table 3.1: List of possible occurrences of parts of a file's path in an issue.

variant of the file path	description	how it can occur in the issue
full file path	-	stack traces, GitHub links to the file
short file path	file name with its parent directory	as an abbreviation for the full file path that appears too lengthy
file name with type	-	mentions of the file
file name without type	-	import of the file, package notation of the file, mentions of just the concept a file represents
permalink	file path appended by a #, that would be followed by a line number in the link	a GitHub feature to link to lines in the code that are embedded and highlighted in the issue

3.4.1 History-based features

I can safely assume that the software engineering in a project follows some process such as Scrum that involves phases and that even in less orchestrated open-source software development the focus of work naturally shifts from time to time.

Despite this intuition, I assume that changes of a file occur in bulk with regard to time. Reasons for this can be that after starting to work on a feature and looking closer at one component of the project, ideas for more features arise or that a file change induced a bug that has to be solved immediately. Just one example that supports this notion can be found in Figure 3.2. Consequently, I assume that the information about how recently a file was edited can influence the prediction about whether a file should be changed. I construct two features that contain the recency of a file change with respect to the issue.

With the operation o I generate a *binary recent file change feature* by the following procedure where o is parameterized by the number of days $d \in \mathbb{N}$:

1. let t_i be the point in time at which the issue i is created and t_f the most recent point in time at which the file f was changed before t_i
2. then Δt is the difference between t_i and t_f in days
3. the feature is active (1) if $\Delta t \leq d$ and inactive (0) otherwise

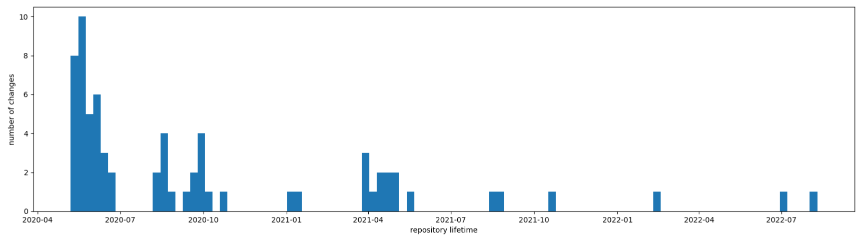


Figure 3.2: Changes of the *server/services/submission/controller/SubmissionController.java* file over the lifetime of *corona-warn-app/cwa-server*. This file is amongst the rather frequently edited files.

Designing a continuous feature that conveys the recency of a file's latest change with respect to an issue, I have to overcome various pitfalls.

- Within one repository, the maximum time for which any file remains untouched, which I call m, increases. This means that the range of values seen during training and testing can vary.
- For different repositories, m can vary a lot not only based on their age but also based how active developers are around the repository.
- Normalizing the time difference between the latest change of a file and the filing of an issue could be a measure to cope with the above problems. Now, if there is just one file that was not edited for a long time, all rather recently edited files, that I am mostly interested in, become indistinguishable. This is equally true when normalizing over the feature values for multiple or one repository or just the files that belong to one issue.

To tackle these problems, I define the operation o that generates the *continuous recent file change feature* by the following procedure:

1. let t_i be the point in time at which the issue i is created, $\mathbf{f}$ are all files that exist at t_i and $\mathbf{t}$ are the respective most recent points in time at which each of the files was last changed
2. all times in $\mathbf{t}$ are rounded up to hours and sorted ascending in $\mathbf{t}_s$
3. according to their position in $\mathbf{t}_s$, files get assigned a rank r where files with the same time get the rank of the highest position of that time

4. ranks are normalized to the range $[0, 1]$
5. now for $f = \mathbf{f}_k$ for a $k \in \{1...|\mathbf{f}|\}$, o yields the normalized rank that was assigned to f

I specifically design this continuous feature to compare how binary and continuous features of the same property differently affect the model performance.

3.4.2 Developer-based features

By this day, programmers are an essential component in software development. Often there is a fixed set of main programmers contributing to a project with different levels of expertise. They occur as authors of issues, editors of files, assignees to proofread pull requests, they give comments in pull requests and issues and are tagged by other programmers, just to name some of their responsibilities.

I assume that the activity of a programmer differs per component of the software project, depending on their task and interest. Thus, getting the signal of activity of a specific programmer anywhere in the project, one can limit the set of files that are probably involved in this activity or will be involved in following activities. One can imagine many possible features in this direction.

Here I aim to make a point that it is worth considering such features and only construct a simple binary feature as an example.

This *author feature* is constructed by an operation o as follows and parameterized by $c \in \mathbb{N}$:

- let a_i be the author of the issue i and c_{a_i} a counter that counts how many times f was changed before to resolve an issue by a_i
- the feature is active (1) if $c_{a_i} \geq c$ and inactive (0) otherwise

The notion behind this feature is that programmers write issues about parts of the code where they are specifically interested or proficient in or that they use frequently. This could be only a small part of the repository such that this feature being active can indicate a preference for the files in this part.

I want to give a supportive example that this feature is worth to investigate further: the three most active issue authors (by number of issues they filed) in the *provectus/kafka-ui* subset of Rütz [Rüt24] filed issues that as a consequence changed 66, 9 and 22 uniquely different files respectively. Among those three sets of files only one file was in common. This can be seen as an indicator that authors of issues tend not to raise issues that involve file changes in other programmers' area of expertise.

3.5 Prior setting for features from semantic embeddings

In Section 3.2.2, I left out how to set the prior for dimensions that are not very meaningful alone. Thus, in this section, I define how to set an informative prior for each dimension of the vector representing the *semantic similarity feature* as well as the *top level directory feature*. To achieve this, I can make use of the importance of each dimension that the embedding model learned already.

For this purpose, I am exploring the real text embedding vectors of issues and files. Firstly, for high dimensional embedding vectors that are created by large neural networks, it is difficult to infer any properties about them up front beyond that they have a unit length. Thus, I turn towards analyzing the embedding vectors empirically. As the representative Figure 3.3 shows, the distribution of values of a single embedding dimension is mostly bell shaped. Besides that, the means and variances of distributions of values for all embedding dimensions vary a lot as Figure 3.4 shows. It is noteworthy that there are dimensions for which their mean or variance are extreme outliers, like the one displayed in Figure 3.3 for the mean.

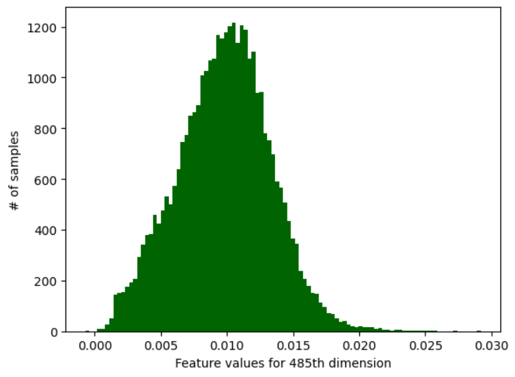
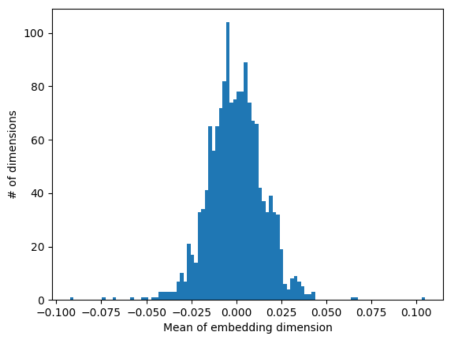
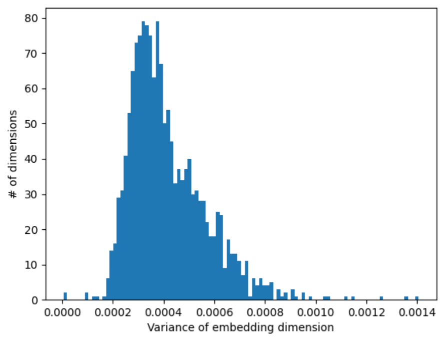


Figure 3.3: Distribution of values of the 485th dimension of OpenAI's *text-embedding-3-small* for 34570 embeddings of file contents from *corona-warn-app/cwa-server*

Now, for simplicity, I assume that $\mathbf{e}_i$ and $\mathbf{e}_f$ as introduced in Section 3.3.2 are multivariate random variables that are independent of each other following the same distribution with expected value $\boldsymbol{\mu}_j$ and variance $\boldsymbol{\sigma}_j{}^2$ for each dimension j and no correlations between dimensions. For $\mathbf{x} = \mathbf{e}_i \odot \mathbf{e}_f$ this gives a per dimension expected value of $\boldsymbol{\mu}_j{}^2$ and variance of $(\boldsymbol{\sigma}_j{}^2 + \boldsymbol{\mu}_j{}^2)^2 - \boldsymbol{\mu}_j{}^4 = \boldsymbol{\sigma}_j{}^4 + 2 \cdot \boldsymbol{\sigma}_j{}^2 \cdot \boldsymbol{\mu}_j{}^2 = \mathbf{z}_j{}^2$ which in practice are obtained from the data.



(a) Sample means for each of the 1536 embedding dimensions of OpenAI's *text-embedding-3-small* from 34570 embeddings of file contents from *corona-warn-app/cwa-server*



(b) Sample variances for each of the 1536 embedding dimensions of OpenAI's *text-embedding-3-small* from 34570 embeddings of file contents from *corona-warn-app/cwa-server*

Figure 3.4: Comparison of all embedding dimensions by per dimension statistics.

From that, the importance which the embedding model implies for each dimension can be derived. Take the embeddings $\mathbf{e}_i$, $\mathbf{e}_f$ and the cosine similarity measure $\sum_{j=1}^{l} \mathbf{e}_{i_j} \cdot \mathbf{e}_{f_j} = \sum_{j=1}^{l} \mathbf{x}_j$ as a ranking function for the relevance of a file with respect to an issue. Let $\mathbf{x}$ be standardized such that $\sum_{j=1}^{l} \frac{\mathbf{x}_j - \boldsymbol{\mu}_j{}^2}{\mathbf{z}_j}$ is obtained. The centering can be neglected by assuming that dimensions are already centered as it does not influence the ranking of files which yields $\sum_{j=1}^{l} \frac{\mathbf{x}_j}{\mathbf{z}_j}$. Thus, from

$$\sum_{j=1}^{l} \frac{\mathbf{x}_j}{\mathbf{z}_j} \cdot \mathbf{z}_j = \sum_{j=1}^{l} \mathbf{x}_j \tag{3.35}$$

it becomes apparent that $\mathbf{z}_j$ is an implicit weight applied per dimension when cosine similarity on a non-standardized $\mathbf{x}$ is used. This is the importance that the embedding model assumes for each dimension. Further, I can infer that this weight is especially large for embedding dimensions with large absolute $\boldsymbol{\mu}_j$ and $\boldsymbol{\sigma}_j{}^2$ which are displayed as outlier dimensions in Figure 3.4.

I showed in 3.35 that ordinary cosine similarity and cosine similarity with per dimension standardization and weighting can be used interchangeably. I will prefer using the latter in the following as it helps to set specific priors for each dimension that can be compared which then becomes equally true for the learned weights per dimension.

As $\mathbf{z}_j$ is the weight determined by the training process of the embedding model, I want to leverage it by setting the mean of the Gaussian prior for each dimension to $\mathbf{z}_j$. I already motivated that a better weight than $\mathbf{z}_j$ should exist for the task I am dealing with. It might be larger and I also do not exclude the possibility that it is negative. At the same time, I believe that a feature dimension that in general is much less important than another dimension as implied by their $\mathbf{z}_j$, will likely not be more important in this specific task as measured by the absolute value of their weights. Therefore, I set the standard deviation of the Gaussian prior distribution to a multiple of $\mathbf{z}_j$ which yields the prior weight distribution presented in Figure 3.5. As a consequence, when determining the prior according to the scheme presented in Section 3.2.2 the parameter $\boldsymbol{\sigma_w}_j$ is fixed by fixing the same factor $f \in \mathbb{R}_+$ for all dimensions of the *semantic similarity feature*.

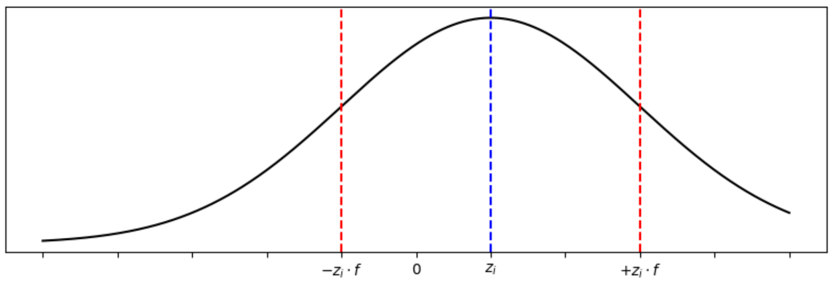


Figure 3.5: Gaussian prior weight distribution for the i-th dimension of the *semantic similarity feature*.

To understand how I set the prior for each dimension of the *top level directory feature*, I start with explaining how I would set it for the *issue semantic feature* (represented by $\mathbf{e}_i$ above), so without the interaction to the top level directory. It is also important to note that I will always use the *semantic similarity* and *top level directory feature* together. That is why, I will pay attention to the relationship between the respective feature vectors of the two features when setting the prior in the following.

I treat the *issue semantic feature* like the *semantic similarity feature* by standardizing each dimension. For this feature, I cannot make any assumption about the correlation of a single dimension to the target value. Consequently, I set a 0 mean for the Gaussian prior of each dimension. Furthermore, I believe that the *semantic similarity feature* is much more important than the *top level directory feature*, as the former contains more specific information about the file. To adequately model my beliefs, I set the Gaussian prior standard deviation for a feature dimension from the *issue semantic feature* to $\mathbf{z}_j \cdot f_d$ where $\mathbf{z}_j \cdot f$ is the Gaussian prior standard deviation for the respective *semantic similarity feature* dimension and $f_d < f$. With this, the factor f_d can represent my belief to which degree the top level directory is less important than the file content. In practice one might not have immediate access to $\mathbf{z}_j$ but knowing the distribution of $\mathbf{e}_i$ as defined above it can be approximated by $\tilde{\mathbf{z}}_j = \sqrt{\boldsymbol{\sigma}_j{}^4 + 2 \cdot \boldsymbol{\sigma}_j{}^2 \cdot \boldsymbol{\mu}_j{}^2}$ which results in the prior distribution shown in Figure 3.6.

Eventually, for a prior that I hypothetically set for a dimension i of the *issue semantic feature*, I set the exact same prior for every i-th dimension of *the top level directory feature.*

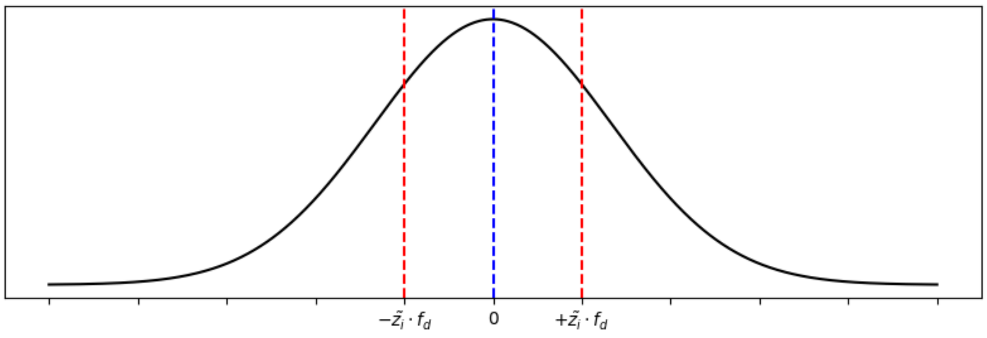


Figure 3.6: Gaussian prior weight distribution for every i-th dimension of the *top level directory feature.*

4 Experiments & Results

4.1 Data

A dataset D for the end-to-end issue-to-code-change task contains samples which each are an issue-to-pull-request pair (i, p). For the file retrieval task in this work, D is transformed into D'. I will refer to a dataset of the form like D as a task dataset, whereas D' is called a retrieval dataset. D'_j is generated from (i_j, p_j) for $j \in \{1...|D|\}$ by:

- let $\mathbf{f}$ be the list of files that are present at the time i_j is created
- then D'_j contains all tuples $(i_j, \mathbf{f}_l, y_l)$ with $l \in \{1...|\mathbf{f}|\}$ and y_l being the target value that is 1 if $\mathbf{f}_l$ is changed in p_j and 0 otherwise

Then D' is obtained by concatenating all D'_j.

4.1.1 Datasets

In this work, models are trained and evaluated on three datasets consisting of different sets of repositories because it allows to examine if the performance of a specific set of features depends on the repository it is used on. Beyond that, some features cannot be constructed for the classic retrieval dataset presented in Rütz [Rüt24].

I will list the three datasets, their specific properties, benefits and report basic overview metrics about their task dataset in Table 4.1 and retrieval dataset in Table 4.2. In Table 4.2 amongst other figures I report how many bits of information can be extracted from the training dataset to inform how the prior is set (see Section 3.2.2). This shows that measuring a maximum of two dataset-specific statistics by default 64-bit floats is appropriate.

CWA retrieval dataset

This dataset stems from *corona-warn-app/cwa-server* that was developed over the course of three years as the backend of the Corona-Warn-App for Germany as a response to the COVID-19 pandemic. It is mainly written in Java. To generate its

Table 4.1: Overview of the task datasets. Explicit links are established in GitHub, implicit links stem from mentions of a link in a comment and insecure links are determined by the heuristic approach.

	CWA	Kafka	Java
# Issue-PR pairs	91	635	772
Avg. Characters in Problem Statement	808	1160	1822
Avg. Gold Files per PR	2.90	2.70	2.20
Avg. Characters in Gold Patch	5860	6194	6147
Explicit links	0.26	0.72	0.17
Implicit links	0.42	0.18	0.83
Insecure links	0.32	0.10	0.00

Table 4.2: Overview of the retrieval datasets.

	CWA		Kafka		Java	
# Intances	25307		511944		76756	
# Files per Instance	278		806		99	
Relevant files	0.011		0.003		0.022	
	java	0.54	java	0.32		
	yaml	0.16	tsx	0.29		
File Types	txt	0.08	ts	0.18	java	1.00
	xml	0.06	yaml	0.07		
	md	0.05	txt	0.03		
	java	0.58	tsx	0.35		
	yaml	0.15	java	0.29		
File Types of Relevant Files	xml	0.09	ts	0.17	java	1.00
	md	0.07	yaml	0.09		
	txt	0.04	xml	0.02		
Bits for prior setting	673		18065		2288	

task dataset the issue-to-pull-request linking procedure described in Rütz [Rüt24] was applied that involves explicitly established links by developers as well as heuristic approaches. The number of issues and files in the repository make it a rather small retrieval dataset which makes it ideal for quick testing and developing a first understanding. On the other hand, results obtained on it are not very representative.

Kafka retrieval dataset

This is a subset of the dataset presented in Rütz [Rüt24] with respect to issues from the *provectus/kafka-ui* repository. It is much larger than the CWA retrieval dataset, primarily due to more issues. It is worth to note that the higher number of files in the repository causes it to have proportionally less relevant files. I specifically picked this repository as it, compared to other repositories from Rütz [Rüt24], has a small number of top level directories that seem to represent self-contained units. I believe that this property is favored by the *top level directory feature* which is an assumption that I want to validate. It is important to mention that this repository contains files from two major programming languages, namely Java and TypeScript.

Java retrieval dataset

This dataset is a post-processed version of the one presented in Rütz [Rüt24] and thus contains issues from 8 different repositories as shown in Table 4.3. It is designed to tackle some of the shortcomings of the above datasets. Table 4.2 shows that the other datasets contain files that are not in the Java programming language. These are mainly files with structured information or formatted text. I suppose that files containing more natural language text could be favored by semantic-text-embedding-based retrieval systems. Controlling for this assumption, this dataset should only contain Java files which are the code files that are of most interest for this work. This also makes it possible to design features that can only be constructed for code files. With more issues across multiple repository, the aim of this dataset is to yield more generalizable models and more robust results. Beyond that, I cannot be certain about the level of noise that is introduced by the partially heuristic data collection approach. Thus, by removing all issue-to-pull-request pairs from the task dataset that were not linked by a programmer, I aim to more robustly assess strength and weaknesses of certain feature sets. Moreover, I reduce the size of the retrieval dataset with feature preprocessing time and model training time in mind. For all samples $(i, f_l, y_l)_{l\in\{1..|\mathbf{f}|\}}$ in the retrieval dataset with a fixed issue i, $|\mathbf{f}|$ files f_l and target variables y_l, only those for which $y = 1$ are included and those for

Table 4.3: Distribution of issues per repository in the Java retrieval dataset.

repository	# of issues
netty/netty	196
provectus/kafka-ui	140
mockito/mockito	123
kestra-io/kestra	111
pinpoint-apm/pinpoint	91
bazelbuild/bazel	64
iluwatar/java-design-patterns	27
square/retrofit	20

which $y = 0$ are randomly undersampled such that only 100 instances with the issue i remain. As this reduces the portion of irrelevant files, better performance outcomes should not come as a surprise.

4.1.2 Splitting

Despite Section 3.2.3 the retrieval task involves time series data such that samples in training, validation and test datasets have to be sorted ascending by time. At the same time, I want to apply cross validation to yield more robust performance metrics. Time series cross validation brings these requirements together. The procedure to generate $k \in \mathbb{N}$ different cross validation datasets is performed on the task dataset D to ensure that a split in D' contains all instances that belong to the same issue.

- assume that the instances in D are sorted ascending by creation time of the issue
- split D into $k + 1$ equally sized chunks $\{D_1 ... D_{k+1}\}$
- the cross validation dataset $c \in \{1...k\}$ contains $\{D_1...D_c\}$ as the training set, the first half of D_{c+1} as the validation set and the second half of D_{c+1} as the test set.

For datasets containing issues from multiple repositories, each per repository subset is split according to the above scheme and then concatenated to yield the splits for the whole dataset. For all experiments in this section, I set $k = 4$ such that the cross validation splits depicted in Figure 4.1 are obtained.

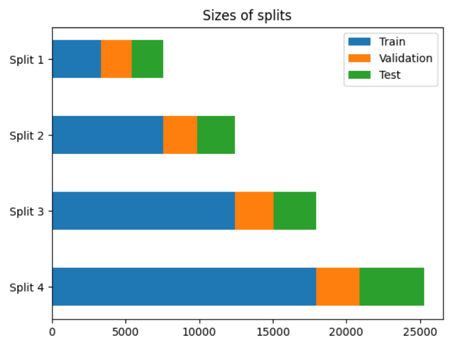


Figure 4.1: Number of retrieval instances per cross-validation dataset and split for *corona-warn-app/cwa-server*

A few pitfalls that come with this approach should be kept in mind, although I do not further investigate them here. The size of the training split in the cross validation datasets increases. As a consequence, I can assume that the model trained on more data naturally generalizes better which can lead to better performance metrics. Beyond that, I suppose that the kind of issues that one can see in a repository evolves over time. Especially, the issues at the beginning of a repository can differ a lot from those in a more mature stage. This can not only cause a very different performance of a model between validation and test split, but in the worst case leads to better performances of non-learning models compared to learning models because training and testing instance are very different.

To get statistics for a dataset as used in Section 3.2.2, they are consistently obtained from D_1 that is part of every training split across all cross-validation datasets. This again can have similar implications like the ones that I just described due to the non-representative distribution of features at the start of the repository.

The validation splits serve the purpose to tune feature as well as model-specific hyperparameters. When tuning parameters, I try to optimize for the AveNDCG score as it better reflects the general ranking ability of a model and where files rank in the top-k.

As metrics to evaluate models, I only report their performance on the test splits for which the scores of performance metrics across all test splits are averaged. To examine and evaluate the test results and trained models beyond the evaluation metrics, I only use test split of the last cross validation dataset (second half of D_{k+1} above) and the respective model.

4.2 Experiments and configuration

In this section, I define the experiments that were run by their feature set and the model. All experiments are run on all three datasets from Section 4.1.1 as far as the chosen feature set allows.

Except for the experiments that use a non-learning model, I train a Bayesian linear probit regression model by the online Byesian Probit Regression algorithm described in Graepel et al. [Gra+10]. More details about the algorithm in the context of our work can be found in Grimm [Gri24]. It follows an online learning approach by iterating over the training data samples, that is why in the described experiments the samples are always randomly shuffled before feeding them into the model to rule out any performance influence that the order of samples might have. I set a Gaussian prior for the weight of each feature vector dimension for which both mean and standard deviation are rounded to one floating point digit.

In line with the data collection approach described in Rütz [Rüt24], I set $k = 10$ to retrieve the top-10 files for each issue. For files that have an expected length of 636 tokens (measured on the CWA retrieval dataset), this should enable us to fit all retrieved files into the context window of recent LLMs (e.g. *text-embedding-3-small* with a context window size of 8191 tokens [Ope24a]).

4.2.1 BM25 on text-based features

I am applying the BM25 algorithm as described in Trotman et al. [TJC12] to the issue description and file contents as the query and the documents of the corpus respectively. To split the strings into keywords, which is a necessary pre-processing step, the queries as well as documents are split at white spaces.

4.2.2 Semantic text-based-features

As motivated in Section 3.3.2, I apply the cosine similarity measure as a non-learning model as well as the above described learning model to the embedding representations of issues and files.

As the model M that generates numerical semantic representations of text, so-called text embeddings, I consistently choose OpenAI's 1536-dimensional *text-embedding-3-small* for which a release note can be found in OpenAI [Ope24b]. A predecessor model is described in Neelakantan et al. [Nee+22] which I will use as a reference. I choose this model for the following reasons:

- easily accessible via an API[12], especially no self-hosting is necessary
- low cost - virtually no cost[13]
- average size of embedding vectors compared to other state-of-the-art models on MTEB [Hug22a; Mue+23], especially half as many dimensions in comparison to OpenAI's slightly more capable *text-embedding-3-large* [Ope24b]
- its predecessor model ranks 4th out of nine models on the Code Information Retrieval Benchmark [Li+24] and second on Code Search Net [Hus+20] examined by Li et al. [Li+24]
- good nDCG@10 score of 51.08 on Benchmarking Information Retrieval [Tha+21] although not state-of-the-art [Hug22b]

In comparison to the BM25 model above, the textual information representing the issue description is enriched by its title and the file content is enriched by its path. To get the numerical representation of the text, I support the model with tags as delimiters to better distinguish between the purposes of each string which is adapted from a common prompt engineering technique for LLMs [Ope].

The issue embedding is obtained by:

$$M(\text{"<issue title>:"} + \text{issue title} + \text{"\textbackslash n\textbackslash n <issue description>"} + \text{issue description}) \tag{4.1}$$

The file embedding is obtained by:

$$M(\text{"<file path>:"} + \text{file path} + \text{"\textbackslash n\textbackslash n <file content>"} + \text{file content}) \tag{4.2}$$

Where + is the string concatenation.

To apply the BLPR model to this feature, prior weight distributions

$$\boldsymbol{w}_i \sim \mathcal{N}(\boldsymbol{\mu}_{\boldsymbol{w}_i}, \boldsymbol{\sigma}_{\boldsymbol{w}_i}^2) = \mathcal{N}(\boldsymbol{\mu}_{\boldsymbol{w}_i}, (\boldsymbol{\mu}_{\boldsymbol{w}_i} \cdot f)^2), \text{(see Section 3.5)} \tag{4.3}$$

for the $i \in \{1...1536\}$ embedding dimension and

$$b \sim \mathcal{N}(\mu_b, \sigma_b^2) \tag{4.4}$$

for the bias term have to be set. $\boldsymbol{\mu}_{\boldsymbol{w}}$ is set as described in Section 3.5. Section 3.2.2 describes how to set σ_b and to calculate μ_b for which Tables 4.5 and 4.6 list the

12 https://platform.openai.com/docs/guides/embeddings
13 due to support for this research by SAP and HPI

Table 4.4: BLPR model performance for the *semantic similarity feature* on the Java retrieval dataset for different values of f.

	Test		Train	
f	AveNDCG	AveR@10	AveNDCG	AveR@10
10	0.80	0.86	0.81	0.88
1	0.68	0.76	0.69	0.75
100	0.79	0.86	0.88	0.95

values used for the experiments. f should be a fixed parameter as well, I want it to be in the right order of magnitude, which so far I am uncertain about. Thus, for $\boldsymbol{\mu_w}$, μ_b and σ_b set as above and $f \in \{1, 10, 100\}$, I train BLPR on the Java retrieval dataset and report its performance in Table 4.4. For $f = 1$ the performance metrics are just a little better than when applying cosine similarity (see Table 4.16), presumably because this setting allows too few deviations from the weights that cosine similarity implies. For $f = 100$ the model performs much better. However, the test metrics for $f = 10$ are as good while metrics on the training data are closer to the ones on the test set which suggests that this model overfits less. Consequently, I will fix $f = 10$.

Similar to the just defined experiment, I run experiments where the file content of code files is reduced to the comments in it - the so called *comment semantic similarity feature*. For those experiments the settings remain as above, besides that the embedding representing the semantic of the file is simply obtained by

$$M(comments). \tag{4.5}$$

Furthermore, I run experiments with both the *standard* and the *comment semantic similarity feature* together, again with $f = 10$.

Extracting comments from Java files is not a straight forward task, due to many possible edge cases. As a simplification I define the function g that extracts only multi-line comments from Java files as:

- let f_c be the content of the file
- get all substrings of f_c that are matched by the regex /*.*?*/ (/* and */ are the comment delimiters; the . here is define as any character including newlines; ? realizes that the substring is closed by the first */)
- clean each multi-line comment my removing /**, /*, *, */, \n from them

Table 4.5: Priors setting for *semantic similarity feature* experiments.

Dataset	Bias prior
Kafka	$\mathcal{N}(-4.8, 1^2)$
Java	$\mathcal{N}(-3.7, 1^2)$
CWA	$\mathcal{N}(-4, 1^2)$

Table 4.6: Priors setting for *comment semantic similarity feature* experiments on the Java retrieval dataset.

Feature set	Bias prior
Comment semantic similarity	$\mathcal{N}(-3.7, 1^2)$
Comment semantic similarity & semantic similarity	$\mathcal{N}(-3.7, 1^2)$

- concatenate all multi-line comments to yield the final comment representation of the file

The *semantic similarity feature* should be the basis for every code retrieval system. I want to investigate if features beyond just semantic representations of the issue's and the file's plain text can be beneficial to such systems. Thus, all following features will as well be accompanied by the *semantic similarity feature* when training a model on them. For the *semantic similarity feature* the prior settings remain as above.

4.2.3 Top level directory feature

I set $n_t = 3$ as the number of important top level directories that are considered and $f_d = 5$ such that $f_d < f$. The prior bias terms are set according to Table 4.7.

Table 4.7: Priors setting for *top level directory feature* experiments.

Dataset	Bias prior
CWA	$\mathcal{N}(-4, 1^2)$
Kafka	$\mathcal{N}(-4.8, 1^2)$

Table 4.8: Performance of binary file path features as a predictor for the target variable in the full Java retrieval dataset. Coverage is the portion of samples where the feature is active.

	Recall	Precision	Coverage	F1 score
full file path	0.048	0.976	0.001	0.091
short file path	0.048	0.976	0.001	0.091
file name with type	0.100	0.677	0.003	0.175
file name without type	0.223	0.501	0.010	0.309
permalink	0.041	1.000	0.001	0.078

Table 4.9: Priors setting for *file name without type feature* experiments.

Dataset	File name without typ prior	Bias prior
CWA	$\mathcal{N}(2.2, 1^2)$	$\mathcal{N}(-4, 1^2)$
Java	$\mathcal{N}(4, 1^2)$	$\mathcal{N}(-3.9, 1^2)$
Kafka	$\mathcal{N}(1, 1^2)$	$\mathcal{N}(-4.9, 1^2)$

4.2.4 File path features

The process to construct the *full file path*, *file name with type* and *permalink features* are straightforward to construct in practice. For the *file name without type feature*, I remove any dots at the beginning of a file name that can exist for hidden files. Then this feature becomes the string until the next dot in the file name such that e.g. *.gitignore* becomes *gitignore*. Moreover, the short file path for paths that are more than one directory deep consists of the file name and its parent directory otherwise it is the the full file path.

I prefer to only run experiments for some of these quite similar features and thus want to select two of them that are particularly promising. For this purpose, I investigate how good of a predictor each of the proposed file path features is for the retrieval task. From the results in Table 4.8, I find that out of all features that can contain directories the *permalink feature* performs best, especially the precision of 1.0 seems to be a promising property. Besides that, the *file name without type feature* stands out as it has the highest coverage and f1 score. Consequently, I only run experiments for the *permalink* and *file name without type features*.

The prior distributions for the weights of those features shown in Tables 4.9 and 4.10 are set as described for binary features in Section 3.2.2.

Table 4.10: Priors setting for *permalink feature* experiments.

Dataset	Permalink prior	Bias prior
CWA	$\mathcal{N}(8.7, 1^2)$	$\mathcal{N}(-4, 1^2)$
Java	$\mathcal{N}(8.5, 1^2)$	$\mathcal{N}(-3.7, 1^2)$
Kafka	$\mathcal{N}(9.5, 1^2)$	$\mathcal{N}(-4.8, 1^2)$

Table 4.11: Priors setting for *binary recent file change feature* experiments.

Dataset	Binary recent file change prior	Bias prior
CWA	$\mathcal{N}(0.3, 1^2)$	$\mathcal{N}(-4.4, 1^2)$
Java	$\mathcal{N}(1.3, 1^2)$	$\mathcal{N}(-3.8, 1^2)$
Kafka	$\mathcal{N}(0.7, 1^2)$	$\mathcal{N}(-5, 1^2)$

4.2.5 Recent file change features

In practice, the essential information about the time a file was last changed at the time of creation of an issue is obtained from the history of the versioning system (git) of the repository. I decide to set $d = 1$ days to parameterize the *binary recent file change feature* for the experiments. The model priors are set as specified in Tables 4.11 and 4.12 where the prior for the continuous feature is set as described for continuous features in Section 3.2.2 with a_1, a_2 determined from 4-quantiles.

4.2.6 Author feature

I set $c = 1$ to parameterize the feature generation process. A pitfall of this feature could be that I can only make use of the issue-pull-request pairs that I have access to, to determine the number of times a file got changed as a consequence of an issue by a specific author. The ground truth for this number might be higher including

Table 4.12: Priors setting for *continuous recent file change feature* experiments.

Dataset	Continuous recent file change prior	Bias prior
Java	$\mathcal{N}(0.9, 1^2)$	$\mathcal{N}(-4.5, 1^2)$
CWA	$\mathcal{N}(0.6, 1^2)$	$\mathcal{N}(-5.4, 1^2)$
Kafka	$\mathcal{N}(0.6, 1^2)$	$\mathcal{N}(-5.9, 1^2)$

Table 4.13: Priors setting for *author feature* experiments.

Dataset	Author prior	Bias prior
CWA	$\mathcal{N}(0.8, 1^2)$	$\mathcal{N}(-4, 1^2)$
Java	$\mathcal{N}(3.6, 1^2)$	$\mathcal{N}(-3.8, 1^2)$
Kafka	$\mathcal{N}(1.5, 1^2)$	$\mathcal{N}(-4.9, 1^2)$

Table 4.14: Model performance for the CWA retrieval dataset.

Model - Feature set	AveNDCG	AveR@10	AveP@10	AAF
BM25 - text-based	0.31	0.23	0.04	0.17
Cosine similarity - semantic similarity	0.54	0.47	0.09	0.36
BLPR - semantic similarity	0.58	0.56	0.11	0.39
BLPR - top level directory	0.58	0.57	0.12	0.42
BLPR - file name	0.61	0.59	0.12	0.47
BLPR - permalink	0.59	0.59	0.12	0.44
BLPR - binary recent file change	0.54	0.52	0.12	0.36
BLPR - continuous recent file change	0.56	0.55	0.12	0.39
BLPR - author	0.45	0.35	0.09	0.25

pull-requests that were created as a response to an issue but were not detected as such by us [Rüt24]. The prior for this feature is specified in Table 4.13.

4.3 Results

The relevant evaluation metrics from Section 3.2.3 for all experiments per dataset are reported in Tables 4.14, 4.15 and 4.16. Experiments for all feature sets were executed for all datasets, except that *comment-based features* were only used for the Java retrieval dataset and that the *top level directory feature* could only be used for the single-repository retrieval datasets. It is important to note that the different datasets seem to pose a differently hard retrieval task as the average of AveR@10 across all experiments for each dataset differs a lot. However, this will not be a focus of my discussion. Besides that, it is not consistent how well a model trained on a certain feature set performs depending on the dataset. Especially, *permalink*, *continuous recent file change* and the *file name feature* (each combined with the *semantic similarity feature*) tend to perform best with respect to the CWA, Kafka and Java retrieval dataset.

Table 4.15: Model performance for the Kafka retrieval dataset.

Model - Feature set	AveNDCG	AveR@10	AveP@10	AAF
BM25 - text-based	0.18	0.03	0.01	0.02
Cosine similarity - semantic similarity	0.34	0.29	0.06	0.19
BLPR - semantic similarity	0.44	0.43	0.08	0.30
BLPR - top level directory	0.44	0.44	0.09	0.31
BLPR - file name	0.40	0.36	0.07	0.24
BLPR - permalink	0.44	0.43	0.08	0.30
BLPR - binary recent file change	0.40	0.41	0.08	0.29
BLPR - continuous recent file change	0.42	0.43	0.09	0.29
BLPR - author	0.43	0.43	0.08	0.29

Table 4.16: Model performance for the Java retrieval dataset.

Model - Feature set	AveNDCG	AveR@10	AveP@10	AAF
BM25 - text-based	0.47	0.49	0.08	0.40
Cosine similarity - semantic similarity	0.65	0.73	0.14	0.62
BLPR - semantic similarity	0.80	0.86	0.17	0.76
BLPR - comment semantic similarity	0.54	0.49	0.08	0.41
BLPR - (comment) semantic similarity	0.77	0.84	0.16	0.75
BLPR - file name	0.80	0.87	0.17	0.76
BLPR - permalink	0.80	0.86	0.17	0.76
BLPR - binary recent file change	0.76	0.86	0.17	0.77
BLPR - continuous recent file change	0.74	0.80	0.15	0.69
BLPR - author	0.78	0.87	0.17	0.79

5 Discussion & Limitations

The performance metrics that I gathered for each dataset and set of features, can be found in Tables 4.14, 4.15 and 4.16. I will examine one feature set at a time and judge its relative performance compared to baseline approaches as well as the properties for each dataset that might have influenced this performance.

The AveNDCG and AveR@10 metrics are highly correlated despite the experimental results. Thus, I can mainly compare the approaches based on their AveR@10 performance. The remaining metrics would be of more interest for downstream code generation tasks. Due to the pre-processing used to generate the Java retrieval dataset, results on it cannot be used to estimate the performance of an end-to-end issue-to-code pipeline. Still, I can point out that despite the AveP@10 metric scores for the two single repository datasets, there is on average only one file forwarded to the generative model. This does not give the generative model the ability to reason about changes in multiple files together. Instead, another retrieval task is posed on it which could severely impact its ability to make the right code change. Besides that, the AAF scores set an upper limit on the portion of issues that can be resolved. In Jimenez et al. [Jim+23] a AAF score of 26% lead a generative model to resolve 2% of the issues. Interpolating, I can assume that a generative model with access to the files retrieved by the presented models (with a maximum AAF score of 44% on the CWA retrieval dataset) might completely resolve around 4% of the issues which falls far behind state-of-the-art agentic approaches [Yan+24].

5.1 Non-learning baselines

5.1.1 BM25

BM25 shows the lowest scores across dataset and metrics by far, possibly due to the inherently different keywords in natural language in comparison to code. This is in line with the observations made in Zhang et al. [Zha+21] by comparing BM25 to post-BERT language models for code search. One reason for the poor performance of my BM25 retrieval in comparison to other work [Jim+23] might be the approach how keywords are determined. In this context Zhang et al. [Zha+21] shows that classic NLP text-pre-processing techniques as well as code-tokenization improve the

Table 5.1: Most important keywords in the BM25 retrieval results for CWA retrieval dataset.

Most important keyword	Count
bug	5
Current	5
[	4
Describe	4
consider	3
lower:	2
|	2
Suggested	2
allowed	2
security	2

retrieval performance of BM25 while Jimenez et al. [Jim+23] uses general-purpose text-tokenizers to generate keywords.

To argue that the presented BM25 approach fails to recognize important keywords from the issue, I examine the keywords for each query that increased the BM25 score for any document the most. Despite Table 5.1 most of those keywords are not specifically connected to the repository or programming and thus fail to represent semantic similarities between issues and files.

5.1.2 Semantic similarity by cosine similarity

Throughout all datasets, metrics at least around double in comparison to BM25 which demonstrates the superiority of semantic text search for this task in comparison to bag-of-words methods.

5.2 Learning baseline

Learning a BLPR model on the *semantic similarity feature* again results in an increase across all datasets and metrics compared to previously discussed approaches, especially there is an increase in AveR@10 of around 10%. As this feature set is contained in (almost) all other feature set for learning models, I will consider this model as a baseline for learning models. I will compare other models to this model by referring to it as Model 5.2. The results indicate that it is possible to fine-tune generalized cosine similarity on text embeddings to the specific task of code retrieval.

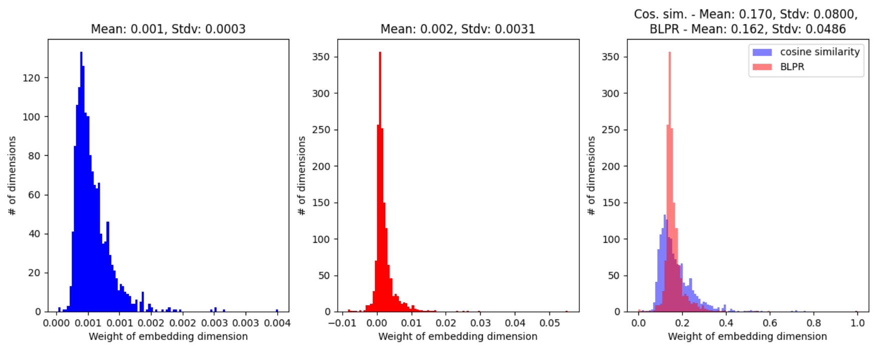


Figure 5.1: Comparison of weight distributions for models on *semantic similarity feature* for the CWA retrieval dataset. For cosine similarity (blue) the implicit weights (see Section 3.5) and for BLPR (red) the learned weights are displayed.

To understand this performance improvement better, I take a look at the implicit per-dimension weights of the model from Section 5.1.2 compared to the weights learned by the model that is discussed in this section in Figure 5.1. It becomes apparent, that the range of learned weights deviates a lot from the default weights. Especially, the negative learned weights indicate that for this task the relationship between a feature vector dimension and the target variable can be inverse compared to the default setting. The results of the ranking task are not affected by scaling all weights by the same factor. Thus, I can scale the weights of both approaches to the range 0 to 1 to better compare them. It shows that the distribution of learned weights has a smaller standard deviation. On the one hand, this mean that dimensions have a more equal learned importance. On the other hand, single dimensions that were assigned a weight that is an extreme outlier in the distribution dominate the model's predictions more.

5.3 Beyond the baselines

The previously analyzed results tend to confirm common knowledge that is in line with the observations made in other work [Zha+21], while in contrast this section evaluates the contribution of less common and novel feature sets for the code retrieval task.

5.3.1 Feature sets with file comments

When only making use of the *comment semantic similarity feature* the performance metrics are just slightly better than those obtained by the BM25 model. Especially, they are worse than when applying simple cosine similarity on the *standard semantic similarity feature*. This performance decrease suggests that some important information from the code file is not considered when reducing it to the multi-line comments contained in it.

I further analyze which exact files were retrieved by this approach. Out of 124 files that were not retrieved by Model 5.2, just 3 are ranked amongst the top-10 files when the *comment semantic similarity feature* is used. I cannot determine a common pattern that suggest why comments seem better suited to represent these exact files. This fact indicates that the two features are not very complementary and that the numerical representation of comments instead of full files is not able to adequately represent code in the code retrieval task. This can as well explain, why a model using both features together performs slightly worse than Model 5.2 as not much information is added while the feature vector size doubles.

The inferiority of a comment representation of files is well displayed by the lower weights of the related feature vector dimensions that were learned by the model using both features in Figure 5.2.

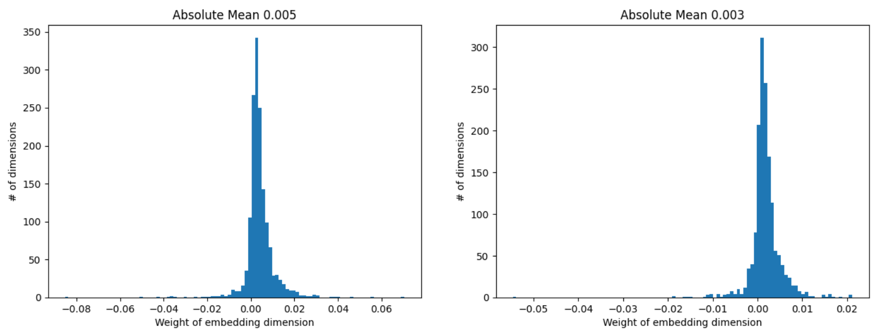


Figure 5.2: Comparison of learned weight distributions for dimensions representing *semantic similarity feature* on the left and dimensions representing *comment semantic similarity feature* on the right.

In contrast, Model 5.2 retrieves 51 files that are not retrieved when only using the *comment semantic similarity feature*. Here some common pitfalls for the comment representation become obvious. Namely, 31 of those files did not include any multi-line comment and most others only contained a license statement, a reference to the author of the file or a single word. However, those files with comments containing seemingly more information, such as method signatures, just slightly fall out of the top-10. Thus, I cannot seriously judge why comments are a worse representation for them.

5.3.2 Top level directory feature

Introducing the *top level directory feature* provides the model with five times larger feature vectors. Nevertheless, the improvement of the model despite the AveR@10 score are minimal with just 1% compared to Model 5.2.

For the high dimensional feature vector representing the *top level directory interaction feature*, it is challenging to reason about how it influences the performance gain. Consequently, I will analyze each part of the feature vector representing a single directory and how it influences the retrieval of files from this directory.

For this purpose, I group the weights that are learned by the model by the directory that they encode in Figures 5.3 and 5.4 for both single repository datasets. Firstly, as expected, absolute weights of the *top level directory feature* dimensions are about 10 times smaller than for the *semantic similarity feature* and weights tend to be distributed more evenly around 0 as assumed by the prior weight distributions. Secondly, one can observe that the feature dimensions encoding a specific directory that contains actual code are the most important by the average absolute weights learned for them. Beyond that, I can report that the uncertainties about the weights are similar across directories.

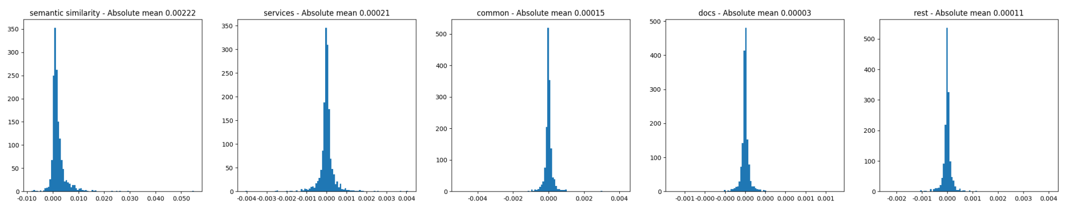


Figure 5.3: Comparison of learned weight distributions for dimensions representing the *semantic similarity feature* and the interaction of the *issue semantic feature* with each of the top level directories for the CWA retrieval dataset.

To showcase the impact of this feature on which files are retrieved by a model, I visualize the distribution of files from a certain directory over the entire ranking of files in Figures 5.5 and 5.6. With this, one is able to validate that this feature contributes to the model's predictions as expected, despite the almost insignificant increase in model performance.

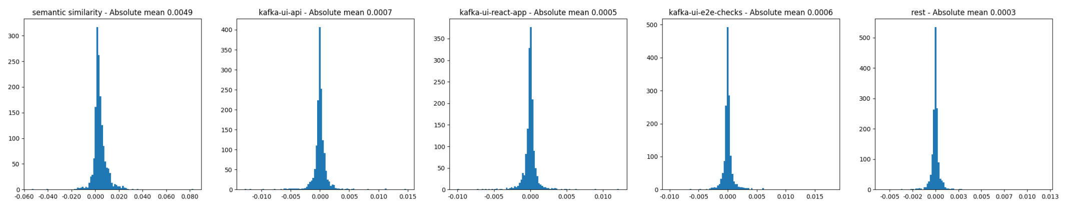


Figure 5.4: Comparison of learned weight distributions for dimensions representing the *semantic similarity feature* and the interaction of the *issue semantic feature* with each of the top level directories for the Kafka retrieval dataset.

In the ranking distribution graphs, a line above the diagonal towards the upper left corner indicates that files in this directory are given rather high retrieval ranks. Thus, for issues for which files in a certain directory have to be edited one would desire that the graph representing files in this directory lies above the diagonal and the respective graphs for other directories should lie below. Still, it has to be considered that issues exist that cause files in different directories to be changed. In Figures 5.5 and 5.6, I can observe the trend that for the model with the *top level directory feature* the line representing files in the relevant directory is corrected towards the upper left and the ones representing files from potentially irrelevant directories are pushed more towards the bottom right. This behavior shows that by introducing this feature, the model's preference for retrieving a file from the correct directory increases. For the Kafka retrieval dataset this behavior is even stronger than for the CWA retrieval dataset. A possible reason for this can be that, by design, the three top level directories for *provectus/kafka-ui* are clearly separated units and pull-requests rarely change files in multiple of those directories. Consequently, the relation between the content of an issue and a certain directory becomes easier to identify.

I want to emphasize the particular properties of the *docs* directory in *corona-warn-app/cwa-server* which mainly contains markdown files. Despite Figure 5.5 the model has a strong preference to retrieve files from this directory which is shown across all issues. This is still the case after introducing the *top level directory feature*. The documentation files in this directory are an exact natural language representation of the code contained in the repository. An explanation for the observed effect could be that the embedding model rather sees two natural language texts as similar and that this property is preserved despite our fine-tuning on natural-language-to-code pairs.

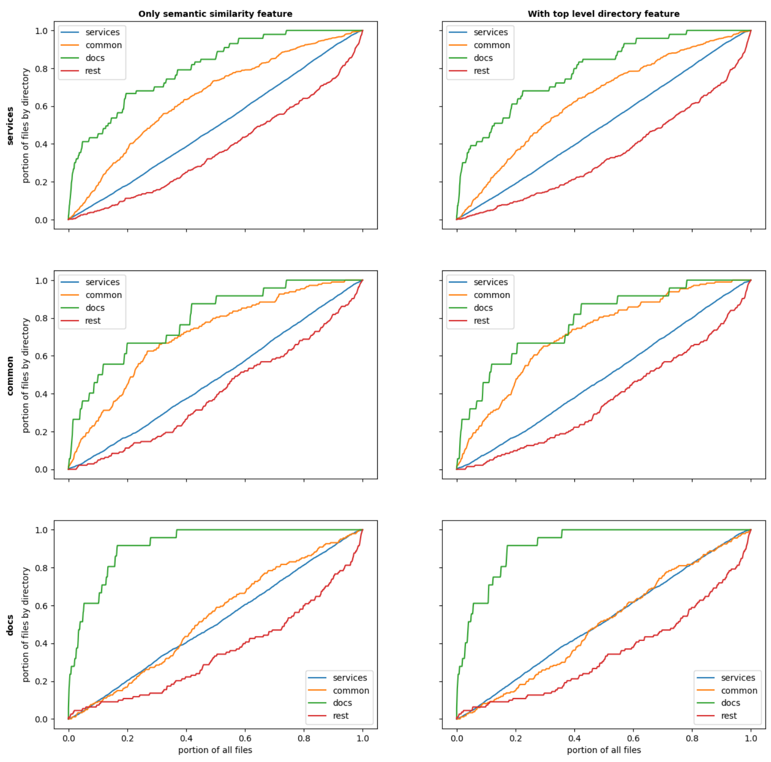


Figure 5.5: Distribution of files by directory over the ranking of all files. The ranks are normalized and averaged for all issues that cause changes in a certain directory (top to bottom). Different feature sets (left, right) for the CWA retrieval dataset are compared. There are no issues in the test set that cause a file in another than the shown directories to be edited.

5.3.3 Feature sets with binary features

In general, across all datasets, the *permalink* and *file name features* mostly bring slight improvements in the AveR@10 score compared to Model 5.2 whereas introducing the *binary recent file change* and *author feature* mostly results in a clearly worse performance which I want to investigate the causes for.

All binary features that are used are designed in a way that if the feature is active, it should give the model a bias towards retrieving more files. The extent of this bias that I also used to set the prior for each binary feature heavily depends on the feature and dataset.

A first reason for that none of these features lead to a great performance gain is that samples where the binary features are active are rare. Besides that, most of the

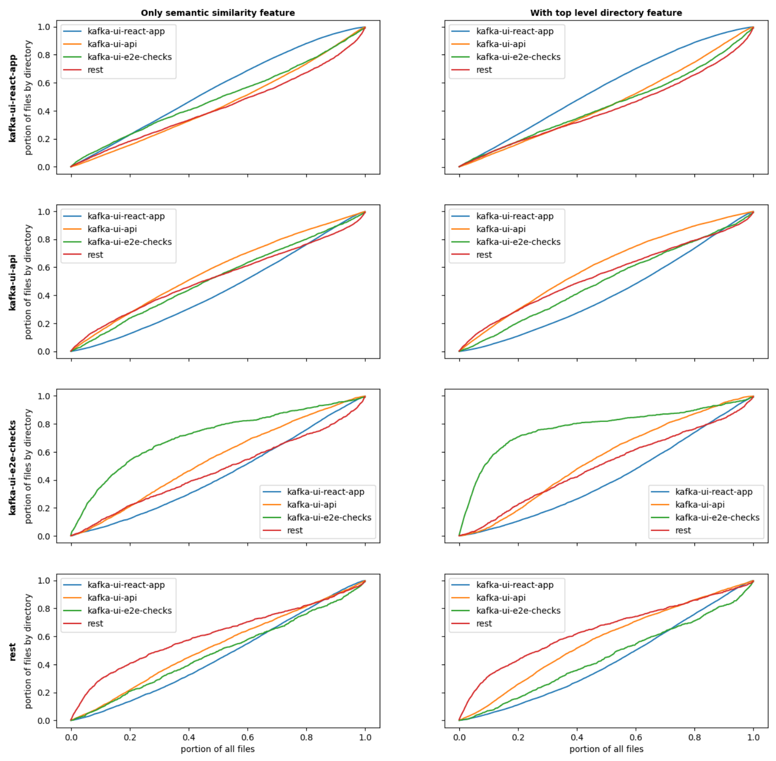


Figure 5.6: Distribution of files by directory over the ranking of all files. The ranks are normalized and averaged for all issues that cause changes in a certain directory (top to bottom). Different feature sets (left, right) for the Kafka retrieval dataset are compared.

relevant files for which the feature is active have already been retrieved by Model 5.2. Thus, no great improvements can be expected as shown in Tables 5.2, 5.3, 5.4 and 5.5 for which probability values are the portion of files for which the specified condition is true.

A second reason for getting worse results, is that the model overestimates the influence of the binary feature. Given a binary feature which receives a positive weight, the files where it is active tend to be ranked higher and other files are pushed down in the ranking. This should happen to a degree that is in line with the probability of retrieving a relevant file given one knows the value of the feature. If this happens to a greater extent, then relevant files where the binary feature is not active will rarely be ranked amongst the top relevant files as the binary feature dominates. This is what can be observed for many of those features in my experiments, as examined in detail for specific features below.

Table 5.2: Performance of the learning baseline with respect to the file name feature.

	CWA	Kafka	Java
P(feature active)	0.01	0.01	0.01
P($y = 1$|feature active)	0.15	0.02	0.55
P($y_{pred} = 1$by learning baseline|feature active, $y = 1$)	0.75	0.64	0.98
# of files not retrieved by learning baseline with feature active	1	5	1

Table 5.3: Performance of the learning baseline with respect to the permalink feature.

	CWA	Kafka	Java
P(feature active)	0.000	0.000	0.001
P($y = 1$|feature active)	1.000	0.667	1.000
P($y_{pred} = 1$by learning baseline|feature active, $y = 1$)	0.000	1.000	1.000
# of files not retrieved by learning baseline with feature active	1	0	0

Table 5.4: Performance of the learning baseline with respect to the binary recent file change feature.

	CWA	Kafka	Java
P(feature active)	0.04	0.02	0.02
P($y = 1$|feature active)	0.01	0.00	0.05
P($y_{pred} = 1$by learning baseline|feature active, $y = 1$)	1.00	0.40	0.86
# of files not retrieved by learning baseline with feature active	0	3	1

Table 5.5: Performance of the learning baseline with respect to the author feature.

	CWA	Kafka	Java
P(feature active)	0.02	0.06	0.01
P($y = 1$|feature active)	0.03	0.01	0.29
P($y_{pred} = 1$by learning baseline|feature active, $y = 1$)	0.67	0.49	0.93
# of files not retrieved by learning baseline with feature active	1	20	1

Table 5.6: Overview of the most frequent file names per retrieval dataset with their number of occurrences.

CWA		Kafka		Java	
application	77	Topic	339	App	46
config	37	Nav	298	Mockito	27
org	11	kafka-ui	271	Topic	17
DiagnosisKey	10	App	171	AppTest	11
private	10	Topics	171	Mock	11
pom	7	topics	167	Flow	10
IO	7	topic	166	Steps	9
README	5	Edit	164	Config	6
SubmissionController	5	Message	157	Not	6
log4j2	5	New	152	Example	6

Next, I want to analyze how those two effects occur for each of the binary features, especially for those that decreased the model's performance.

The *permalink feature* never leads to decreasing metric scores which is related to the high precision of this feature alone as a predictor (see Table 4.8), but as the file path is already part of the code embedding and this feature is rarely active, it only enables to retrieve minimally more relevant files.

The *file name feature* improves the model's performance on the Java and CWA retrieval datasets. Reasons for the comparatively far worse performance on the Kafka retrieval dataset can be that for this dataset the ten file names that occur most often in issues are mostly generic words as Table 5.6 shows, whereas for CWA some repository specific keywords can be found.

The influence of the *author feature* reaches from a vast decrease for the CWA dataset to a slight improvement for the Java dataset. The reason for this is that the model overestimates the influence of an active author feature for retrieving a file.

To further investigate this effect, I measure a model's preference for retrieving more files based on an active binary feature b as the proportion:

$$p_M = \frac{\mathrm{P}(y_M = 1|b = 1)}{\mathrm{P}(y_M = 1|b = 0)} \tag{5.1}$$

with y_M as the predicted labels by a model M. There is a ground truth value p for this proportion that is obtained by using ground truth target labels y. For a perfectly accurate model M^*, it would hold that $p_{M^*} = p$. By retrieving the top-k files for the datasets in this work, there are always more files retrieved than are relevant. Still, I assume that a model M' with a recall of 100% under these circumstances would mostly preserve this relation such that $p_{M'} \approx p$ – acknowledging that this is not possible for large $\mathrm{P}(y = 1|b = 1)$. This essentially means that the model is calibrated with respect to the binary feature.

However, Tables 5.7 and 5.8 show that a model does not necessarily have to contain this calibration to be perfectly accurate. For this purpose, I define the model M_{pa} which is perfect but agnostic to the binary feature. Thus, I let it retrieve all relevant files and randomly choose the remaining files that it has left to retrieve. It is shown that such a model also develops a bias with respect to the binary feature (i.e. $p_{M_{pa}} \neq 1.0$).

Now, for setting the prior for a binary feature, (see Section 3.2.2) I assumed for simplicity that p_M is close to 1.0 for a model that does not use the binary feature. Meaning, it does not develop a preference for retrieving files with respect to the binary feature. However, in practice, the reference model M_r (Model 5.2) already captures a good part of p (see Table 5.7). Thus, for a model M with access to the *author feature* the means of the posterior weights for this feature should become smaller than those of the prior weights. One can see in 5.7 that this is the case for the datasets where the *author feature* did not lead to a vast performance decrease. However, the posterior mean is higher than the prior mean for the CWA dataset. Alongside the rather high posterior standard deviation (uncertainty) this implies that the prior weight distribution still dominates over the data in the CWA dataset. This leads to overestimating the influence of the binary feature described by $p_M >> p$ in Table 5.7.

The *binary recent file change feature* also decreased the model's performance on all datasets because there was not much room for improvement (see Table 5.4). Similar to the *author feature* one can observe in Table 5.8 that p is especially overestimated (which leads to a poor model performance) for the datasets where learning from the data did not yield a much smaller posterior mean compared to the prior mean (see Figure 5.8).

In summary, introducing a binary feature clearly gives an advantage for the files where the feature is active, but for most datasets and binary features this advantage is not properly balanced in comparison with the immediate disadvantages, meaning reducing the rank of actually relevant files. This effect can be observed in detail for the *author* and *binary recent file change features* in Appendix B.

Table 5.7: Retrieval of files with an active or inactive author feature based on the chosen feature set, model and retrieval dataset.

	CWA	Kafka	Java
$P(y_{M_{pa}} = 1 \| b = 1)$	0.050	0.018	0.352
$P(y_{M_{pa}} = 1 \| b = 0)$	0.020	0.009	0.098
$p_{M_{pa}}$	2.543	1.962	3.577
$P(y = 1 \| b = 1)$	0.034	0.010	0.292
$P(y = 1 \| b = 0)$	0.003	0.002	0.014
p	9.954	6.668	20.289
$P(y_{M_r} = 1 \| b = 1)$	0.080	0.028	0.562
$P(y_{M_r} = 1 \| b = 0)$	0.019	0.008	0.097
p_{M_r}	4.197	3.439	5.793
$P(y_M = 1 \| b = 1)$	0.368	0.074	1.000
$P(y_M = 1 \| b = 0)$	0.013	0.006	0.094
p_M	27.459	13.282	10.598

Table 5.8: Retrieval of files with an active or inactive binary recent fiel change feature based on the chosen feature set, model and retrieval dataset.

	CWA	Kafka	Java
$P(y_{M_{pa}} = 1 \| b = 1)$	0.028	0.011	0.135
$P(y_{M_{pa}} = 1 \| b = 0)$	0.020	0.009	0.099
$p_{M_{pa}}$	1.410	1.208	1.353
$P(y = 1 \| b = 1)$	0.012	0.004	0.054
$P(y = 1 \| b = 0)$	0.004	0.002	0.015
p	3.221	1.994	3.484
$P(y_{M_r} = 1 \| b = 1)$	0.030	0.030	0.246
$P(y_{M_r} = 1 \| b = 0)$	0.020	0.009	0.097
p_{M_r}	1.516	3.322	2.525
$P(y_M = 1 \| b = 1)$	0.255	0.095	0.477
$P(y_M = 1 \| b = 0)$	0.011	0.008	0.094
p_M	22.548	12.270	5.099

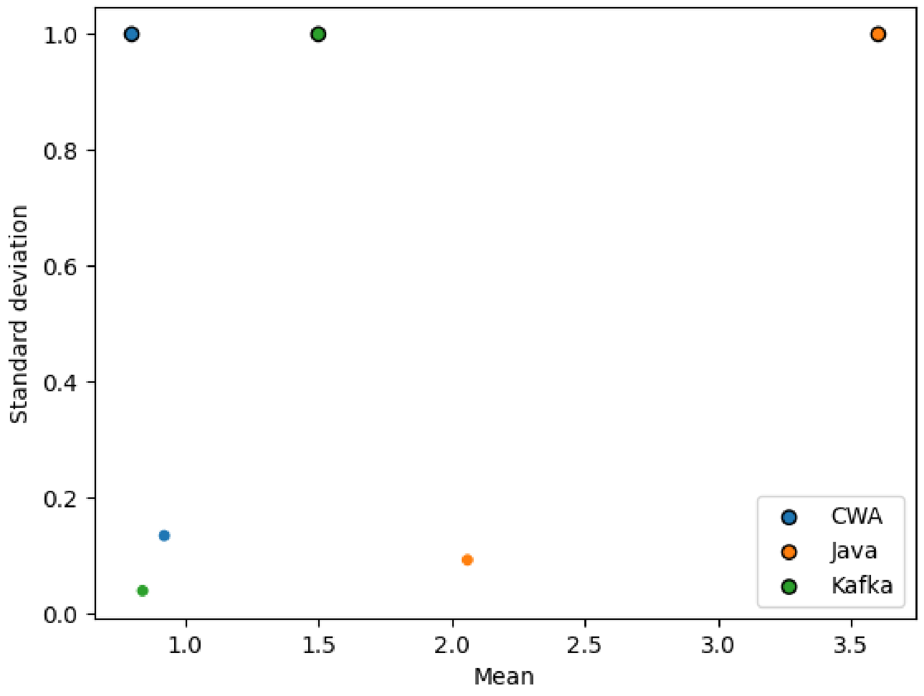


Figure 5.7: Comparison of prior (circle with black edge) and posterior weight distributions for the *author feature* for all retrieval datasets.

5.3.4 Continuous feature

As stated, there are only a few files that are not yet retrieved by Model 5.2 that have an active binary feature, such that one cannot expect much of an improvement by using this kind of feature. This motivates the use of continuous pendants of the binary features that are more nuanced in a way that there are more instances whose relevance score can be positively tweaked in accordance with that feature. As such a continuous feature I investiage *continuous recent file change*.

Analogous to the binary features, a good way to understand why this feature did not benefit the AveR@10 score of any dataset, is to look at the portion of retrieved files p_M given a model M depending on the value c of the continuous feature. p_M is defined by the function:

$$p_M = \mathrm{P}(y_M = 1|c) \tag{5.2}$$

Again, for a model that is well calibrated with respect to the continuous feature, it

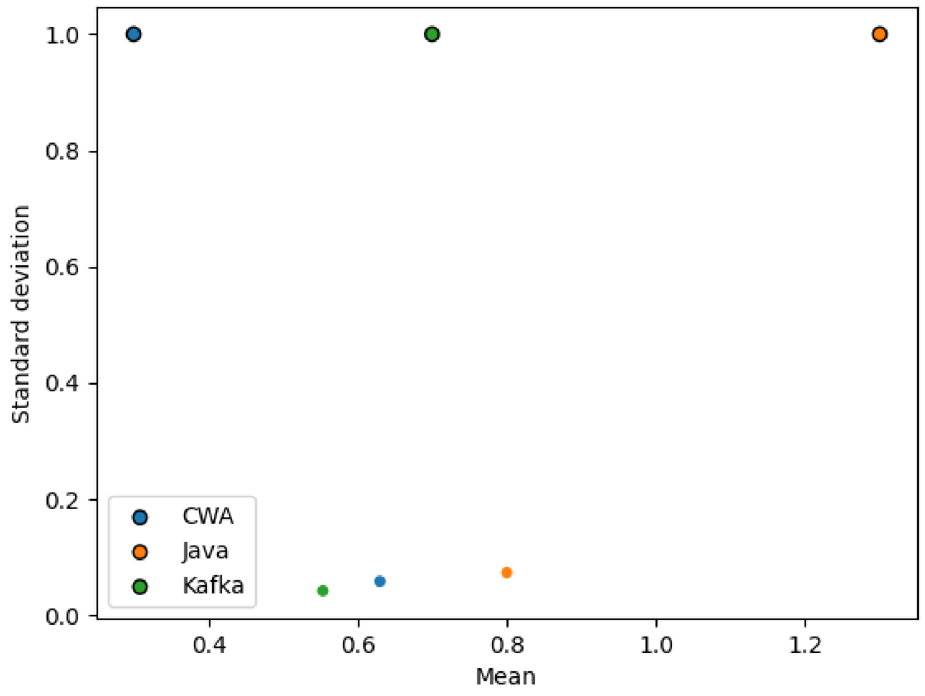


Figure 5.8: Comparison of prior (circle with black edge) and posterior weight distributions for the *binary recent file change feature* for all retrieval datasets.

would be desired that the portion of files that is retrieved for different c is maintained compared to the ground truth. This can be formalized as $p_M \approx a \cdot p$ with a factor $a \in \mathbb{R}, a > 1$ and $p = \mathrm{P}(y = 1|c)$. Depending on the dataset, a can be determined by

$$\frac{\text{fixed \# of retrieved files depending on } k}{\text{\# of relevant files}} \tag{5.3}$$

where the top-k files get retrieved.

I define p_M, p_{M_r} and $p_{M_{pa}}$ analogous to the binary features above.

In Figure 5.9 one can note by the graphs for p that indeed, more recently edited files are related to a higher probability of being relevant. Similar to the binary features, I notice that for all datasets even when not using the *continuous recent file change feature*, if not perfectly, the trend of this relationship is already captured in

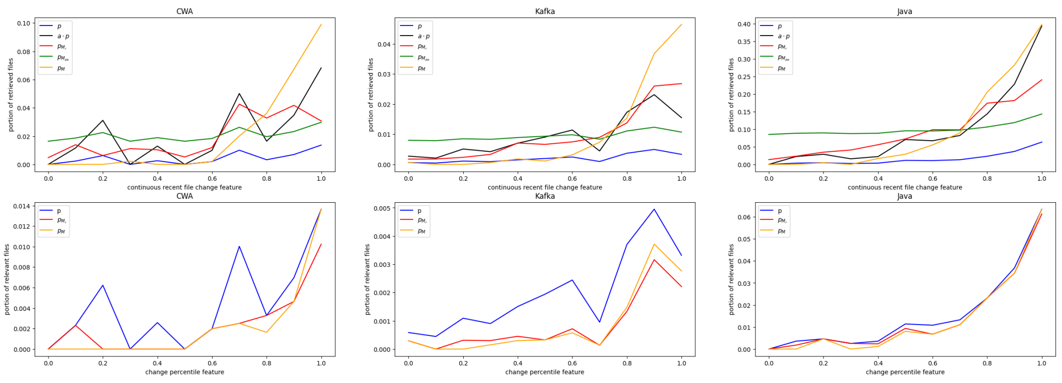


Figure 5.9: Retrieval of files depending on the value of the *continuous recent file change feature* based on the chosen feature set, model and retrieval dataset. The first row displays the portion of retrieved files whereas the second row shows the portion of actually found relevant files. To obtain the values for the graphs the continuous feature was normalized to the range $[0, 1]$ and binned into bins of size 0.1.

the p_{M_r} graphs. Consequently, there seems to be not much room for improvement to align p_{M_r} to the desired $a \cdot p$ graphs.

One can see that all models that are trained on the *continuous recent change feature* have a stronger bias towards more recent files such that in the extremes (for non-recent files) even less files than are relevant get retrieved. This leads to far worse performance metrics shown in 4.14 and 4.16 compared to the baseline Model 5.2. Figure 5.10 shows that for all datasets, the posterior mean of this feature's weight is much smaller than the prior mean which indicates that the prior overestimated the influence of this feature throughout. Nonetheless, the model is quite certain about the weights it eventually learned. As a result, I had to rethink the prior setting procedure in Section 3.2.2 to yield smaller mean priors to prevent the models from developing this bias for recent files. Lastly, the fact that the performance loss on the Kafka retrieval dataset with this feature is negligibly small can mostly be attributed to the generally bad performance on this dataset such that there were many recently edited files left to retrieve as the second row of plots in Figure 5.9 shows.

5.4 Limitations

For the Bayesian approach that I pursued in this work, not only a specific feature set but also the manner in which its prior weights are set influence a model's performance. Thus, statements about the performance of a model only have limited

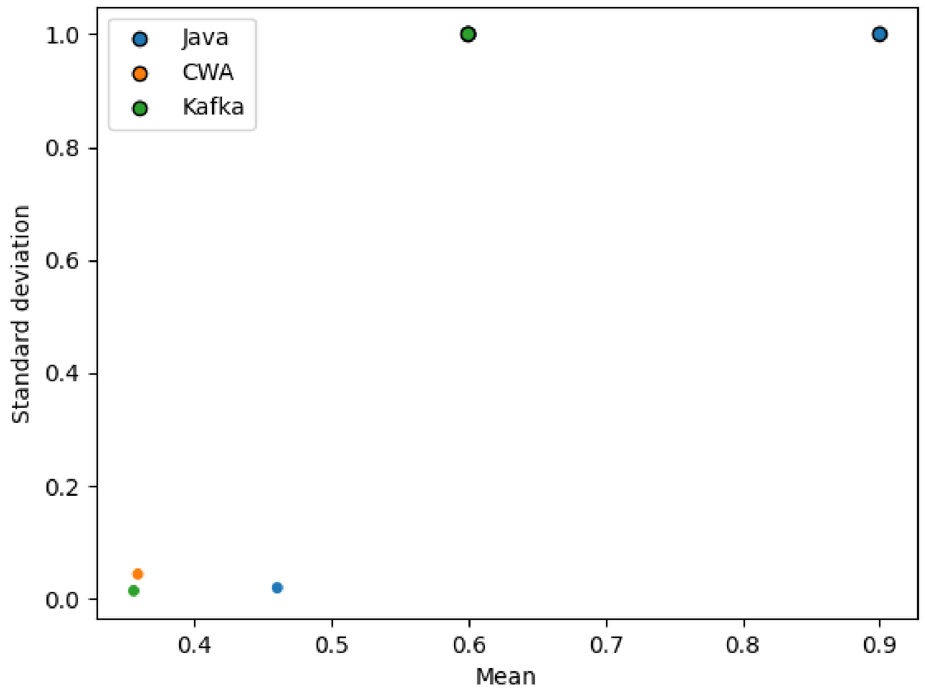


Figure 5.10: Comparison of prior (circle with black edge) and posterior weight distributions for the *continuous recent file change feature* for all retrieval datasets. The prior is the same for CWA and Kafka dataset.

validity for a feature set alone, but also judge the process how the prior was set which was often done with respect to a specific dataset not from common prior knowledge alone.

Besides that, the retrieval datasets that I used do not allow drawing generalizable conclusions from the evaluation of models on them. The diverging performance of same feature sets on the CWA and Kafka datasets shows that there are properties of repositories that I did not investigate deeper and did not control for that influence the retrieval performance. Much of this could be related to how rigorous the software engineering workflow is that is enforced around a repository which could for example include how rigorously issues are described. Although the Java retrieval dataset contains a broader variety of repositories, it is biased towards relevant files and weights some repositories more than others which is a known problem in other coding benchmarks as well [Jim+23].

Another limitation is that in this work used just one embedding model to obtain semantic embeddings. As one could observe the semantic similarity feature dominates the retrieval performance, thus the obtained results only hold with respect to this specific embedding model. In addition, it is difficult to reason about this embedding model because only a few details about it are published.

Most of the specific features that I designed are represented as binary features that might have a vanishing effect compared to features that are represented by high-dimensional embeddings. I consider binary features as not very sophisticated, such that no general conclusions about a feature (e.g. making use of who the author of an issue is) can be drawn as better suitable representations can be found.

Lastly, the BLPR model only allows representing linear relationships between feature vector dimensions and the target value. As most dimensions of the feature vector are obtained from high-dimensional embeddings that stem from a large neural network model, this assumed relationship might be too simple to adequately leverage the semantic representation capabilities of the embeddings.

6 Conclusions & Outlook

In this work I investigated which features can be used to represent issues and files in the context of a code retrieval task and how each feature set and specifically modelled prior knowledge influence the performance in this task.

For this purpose, I leveraged pre-existing text embedding models that are capable of representing the semantic of text as high-dimensional numerical vectors. I could not only show that this seems to yield a better representation of issues and files for the task at hand than bag-of-words approaches, but also that the raw representations obtained from embeddings can be tuned to this task by a simple linear model.

The quality of these semantic representations is also shown by the fact that combining further features with them mostly did not increase the performance of models for the retrieval task and the maximum increase in the average recall@10 was kept to 3%. A reason for this is that much of the expected contribution of a further feature, was already captured by the semantic representation.

I showed, how specific feature sets relate to properties of certain repositories such that I can conclude that selecting as well as constructing features with respect to a single repository can be beneficial in contrast to using a generic feature set for all repositories.

Additionally, I want to emphasize the *permalink feature*, which itself is a close to 100% precision predictor for relevant files, consistently lead to better retrieval performances. This shows how powerful it can be to leverage special features obtained from the software that is used to coordinate the software development around a repository.

While current state-of-the-art agentic solutions like Yang et al. [Yan+24] solve above 12% of all issues completely, such results are much harder to achieve by the two-step approach presented in this work. As a result, an important next step would be to investigate how a retrieval tool like I presented can be used by agentic approaches – potentially alongside the file search tools that are handed to LLM agents in Yang et al. [Yan+24].

Another main point that has to be tackled in future work, is to evaluate the presented feature sets on larger, broader and less noisy datasets. This includes more programming languages and repositories. Some first datasets for this purpose could be CoIR from Li et al. [Li+24] or SWE-bench from Jimenez et al. [Jim+23].

As the *top level directory feature* showed a promising effect on which files are retrieved but failed to significantly improve a model's retrieval performance, it should be further investigated. Its potential ability to decrease the search space of relevant files to a selection of directories seems promising to me. For this, I would propose a multi-stage retrieval approach that at first narrows the search space down to a directory and then selects the files. For this purpose, it could be valuable to semantically represent a directory as the sum of its contained files, opposed to just its name.

In addition, a broader variety of embedding models should be applied to this task. I assume that the 1536-dimensional embeddings are not favored by the limited data that I had access to in this work. Promisingly, OpenAI [Ope24b] suggests that much smaller embeddings can be even more capable.

Eventually, I assumed in this work that files are single independent units in a code base. This is certainly not the case as they heavily depend on other objects that are imported from the same code base or external libraries, use values from configuration files or have a natural language pendant in a documentation. A first idea to build up on my work and reflect those relationships would be to aggregate the semantic representation of all imported files and using it as another feature of the file. In a more sophisticated approach, one could exploit the code base as a whole graph of related files and view the retrieval task as node prediction by a graph neural network for which the approach presented in Dong et al. [Don+24] could be an inspiration.

Bibliography

[Ala18] Jay Alammar. *The Illustrated BERT, ELMo, and co. (How NLP Cracked Transfer Learning).* Dec. 2018. URL: http://jalammar.github.io/illustrated-bert/ (visited on 09/05/2024) (see page 16).

[Che+21] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. *Evaluating Large Language Models Trained on Code.* July 2021. DOI: 10.48550/arXiv.2107.03374. URL: http://arxiv.org/abs/2107.03374 (visited on 08/31/2024) (see page 1).

[Don+24] Jialin Dong, Bahare Fatemi, Bryan Perozzi, Lin F. Yang, and Anton Tsitsulin. *Don't Forget to Connect! Improving RAG with Graph-based Reranking.* May 2024. DOI: 10.48550/arXiv.2405.18414. URL: http://arxiv.org/abs/2405.18414 (visited on 08/31/2024) (see page 62).

[Elo+23] Tyna Eloundou, Sam Manning, Pamela Mishkin, and Daniel Rock. *GPTs are GPTs: An Early Look at the Labor Market Impact Potential of Large Language Models.* Aug. 2023. DOI: 10.48550/arXiv.2303.10130. URL: http://arxiv.org/abs/2303.10130 (visited on 08/31/2024) (see page 1).

[Gra+10] Thore Graepel, Joaquin Quiñonero Candela, Thomas Borchert, and Ralf Herbrich. **Web-Scale Bayesian Click-Through Rate Prediction for Sponsored Search Advertising in Microsoft's Bing Search Engine**. In: *Proceedings of the 27th International Conference on Machine Learning ICML 2010, Invited Applications Track (unreviewed, to appear).* June 2010 (see page 34).

[Gri24] Nico Grimm. **Machine Learning based Approaches to Semantic Code Retrieval**. Bachelor's Thesis. Hasso Plattner Institut (Universität Potsdam), Sept. 2024 (see page 34).

[Gün+24] Michael Günther, Jackmin Ong, Isabelle Mohr, Alaeddine Abdessalem, Tanguy Abel, Mohammad Kalim Akram, Susana Guzman, Georgios Mastrapas, Saba Sturua, Bo Wang, Maximilian Werk, Nan Wang, and Han Xiao. *Jina Embeddings 2: 8192-Token General-Purpose Text Embeddings for Long Documents.* Feb. 2024. DOI: 10.48550/arXiv.2310.19923. URL: http://arxiv.org/abs/2310.19923 (visited on 09/01/2024) (see page 16).

[Hug22a] Huggingface. *Overall MTEB English leaderboard.* Sept. 2022. URL: https://huggingface.co/spaces/mteb/leaderboard (visited on 09/01/2024) (see page 35).

[Hug22b] Huggingface. *Retrieval English leaderboard.* Sept. 2022. URL: https://huggingface.co/spaces/mteb/leaderboard (visited on 09/06/2024) (see page 35).

[Hus+20] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. *CodeSearchNet Challenge: Evaluating the State of Semantic Code Search.* June 2020. DOI: 10.48550/arXiv.1909.09436. URL: http://arxiv.org/abs/1909.09436 (visited on 08/24/2024) (see pages 17, 35).

[Iza+22] Gautier Izacard, Mathilde Caron, Lucas Hosseini, Sebastian Riedel, Piotr Bojanowski, Armand Joulin, and Edouard Grave. **Unsupervised Dense Information Retrieval with Contrastive Learning**. *Transactions on Machine Learning Research* (Aug. 2022) (see page 16).

[Jim+23] Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R. Narasimhan. **SWE-bench: Can Language Models Resolve Real-world Github Issues?** In: Oct. 2023. URL: https://openreview.net/forum?id=VTF8yNQM66 (visited on 09/01/2024) (see pages 1–3, 5, 15, 43, 44, 58, 61).

[Kal22] Eirini Kalliamvakou. *Research: quantifying GitHub Copilot's impact on developer productivity and happiness.* Sept. 2022. URL: https://github.blog/news-insights/research/research-quantifying-github-copilots-impact-on-developer-productivity-and-happiness/ (visited on 08/31/2024) (see page 1).

[Kam23] Gregory Kamradt. *Needle In A Haystack - Pressure Testing LLMs.* Nov. 2023. URL: https://github.com/gkamradt/LLMTest_NeedleInAHaystack (visited on 08/30/2024) (see pages 3, 5).

[Kar+20] Vladimir Karpukhin, Barlas Oguz, Sewon Min, Patrick Lewis, Ledell Wu, Sergey Edunov, Danqi Chen, and Wen-tau Yih. **Dense Passage Retrieval for Open-Domain Question Answering**. In: *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP).* Nov. 2020 (see pages 5, 15).

[Lee+24] Jinhyuk Lee, Zhuyun Dai, Xiaoqi Ren, Blair Chen, Daniel Cer, Jeremy R. Cole, Kai Hui, Michael Boratko, Rajvi Kapadia, Wen Ding, Yi Luan, Sai Meher Karthik Duddu, Gustavo Hernandez Abrego, Weiqiang Shi, Nithi Gupta, Aditya Kusupati, Prateek Jain, Siddhartha Reddy Jonnalagadda, Ming-Wei Chang, and Iftekhar Naim. *Gecko: Versatile Text Embeddings Distilled from Large Language Models*. Mar. 2024. DOI: 10.48550/arXiv.2403.20327. URL: http://arxiv.org/abs/2403.20327 (visited on 08/11/2024) (see pages 16, 17).

[Lew+20] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. **Retrieval-augmented generation for knowledge-intensive NLP tasks**. In: *Proceedings of the 34th International Conference on Neural Information Processing Systems*. NIPS '20. Red Hook, NY, USA: Curran Associates Inc., Dec. 2020, 9459–9474. ISBN: 978-1-71382-954-6. (Visited on 09/01/2024) (see page 3).

[Li+24] Xiangyang Li, Kuicai Dong, Yi Quan Lee, Wei Xia, Yichun Yin, Hao Zhang, Yong Liu, Yasheng Wang, and Ruiming Tang. *CoIR: A Comprehensive Benchmark for Code Information Retrieval Models*. July 2024. DOI: 10.48550/arXiv.2407.02883. URL: http://arxiv.org/abs/2407.02883 (visited on 08/24/2024) (see pages 17, 35, 61).

[Mue+23] Niklas Muennighoff, Nouamane Tazi, Loic Magne, and Nils Reimers. **MTEB: Massive Text Embedding Benchmark**. In: *Proceedings of the 17th Conference of the European Chapter of the Association for Computational Linguistics*. May 2023 (see page 35).

[Nee+22] Arvind Neelakantan, Tao Xu, Raul Puri, Alec Radford, Jesse Michael Han, Jerry Tworek, Qiming Yuan, Nikolas Tezak, Jong Wook Kim, Chris Hallacy, Johannes Heidecke, Pranav Shyam, Boris Power, Tyna Eloundou Nekoul, Girish Sastry, Gretchen Krueger, David Schnurr, Felipe Petroski Such, Kenny Hsu, Madeleine Thompson, Tabarak Khan, Toki Sherbakov, Joanne Jang, Peter Welinder, and Lilian Weng. *Text and Code Embeddings by Contrastive Pre-Training*. Jan. 2022. DOI: 10.48550/arXiv.2201.10005. URL: http://arxiv.org/abs/2201.10005 (visited on 08/11/2024) (see pages 1, 16, 17, 34).

[Ni+22] Jianmo Ni, Chen Qu, Jing Lu, Zhuyun Dai, Gustavo Hernandez Abrego, Ji Ma, Vincent Zhao, Yi Luan, Keith Hall, Ming-Wei Chang, and Yinfei Yang. **Large Dual Encoders Are Generalizable Retrievers**. In: *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*. Ed. by Yoav Goldberg, Zornitsa Kozareva, and Yue Zhang. Abu Dhabi, United Arab Emirates: Association for Computational Linguistics, Dec. 2022, 9844–9855. DOI: 10.18653/v1/2022.emnlp-main.669. URL: https://aclanthology.org/2022.emnlp-main.669 (visited on 09/01/2024) (see page 17).

[OLV19] Aaron van den Oord, Yazhe Li, and Oriol Vinyals. *Representation Learning with Contrastive Predictive Coding*. Jan. 2019. DOI: 10.48550/arXiv.1807.03748. URL: http://arxiv.org/abs/1807.03748 (visited on 08/24/2024) (see page 16).

[Ope] OpenAI. *Tactic: Use delimiters to clearly indicate distinct parts of the input*. URL: https://platform.openai.com/docs/guides/prompt-engineering/tactic-use-delimiters-to-clearly-indicate-distinct-parts-of-the-input (visited on 09/01/2024) (see page 35).

[Ope24a] OpenAI. *Embedding models*. Jan. 2024. URL: https://platform.openai.com/docs/guides/embeddings/embedding-models (visited on 09/01/2024) (see page 34).

[Ope24b] OpenAI. *New embedding models and API updates*. Jan. 2024. URL: https://openai.com/index/new-embedding-models-and-api-updates/ (visited on 08/13/2024) (see pages 34, 35, 62).

[Rob+94] Stephen E. Robertson, Steve Walker, Susan Jones, Micheline Hancock-Beaulieu, and Mike Gatford. **Okapi at TREC-3**. In: *Proceedings of The Third Text REtrieval Conference*. Nov. 1994 (see page 15).

[Rüt24] Max Rütz. **Extracting Problem-Solution Couplings from GitHub Repositories for Enhanced Code Retrieval**. Bachelor's Thesis. Hasso Plattner Institut (Universität Potsdam), Aug. 2024 (see pages 2, 20, 21, 24, 29, 31, 34, 40).

[RZ09] Stephen Robertson and Hugo Zaragoza. **The Probabilistic Relevance Framework: BM25 and Beyond**. *Foundations and Trends® in Information Retrieval* 3 (Dec. 2009) (see page 15).

[SK22] Ted Sanders and Logan Kilpatrick. *Classification using embeddings | OpenAI Cookbook*. July 2022. URL: https://cookbook.openai.com/examples/classification_using_embeddings (visited on 08/31/2024) (see page 1).

[Smi+23] Greg Smith, Michael Bateman, Remy Gillet, and Eystein Thanisch. *Environmental Impact of Large Language Models*. Aug. 2023. URL: https://www.cutter.com/article/environmental-impact-large-language-models (see page 1).

[Sta23] Stack Overflow. *Stack Overflow Developer Survey 2023*. 2023. URL: https://survey.stackoverflow.co/2023/?utm_source=social-share&utm_medium=social&utm_campaign=dev-survey-2023 (visited on 08/31/2024) (see page 1).

[Tha+21] Nandan Thakur, Nils Reimers, Andreas Rücklé, Abhishek Srivastava, and Iryna Gurevych. **BEIR: A Heterogeneous Benchmark for Zero-shot Evaluation of Information Retrieval Models**. In: *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 2)*. Aug. 2021 (see pages 17, 35).

[TJC12] Andrew Trotman, Xiangfei Jia, and Matt Crane. **Towards an Efficient and Effective Search Engine.** In: *SIGIR 2012 Workshop on Open Source Information Retrieval*. 2012 (see page 34).

[Vas+17] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. **Attention is All you Need**. In: *Advances in Neural Information Processing Systems*. Vol. 30. Curran Associates, Inc., 2017. URL: https://papers.nips.cc/paper_files/paper/2017/hash/3f5ee243547dee91fbd053c1c4a845aa-Abstract.html (visited on 09/01/2024) (see pages 3, 16).

[Yan+24] John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. *SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering*. May 2024. DOI: 10.48550/arXiv.2405.15793. URL: http://arxiv.org/abs/2405.15793 (visited on 08/28/2024) (see pages 1, 43, 61).

[Zha+21] Xinyu Zhang, Ji Xin, Andrew Yates, and Jimmy Lin. **Bag-of-Words Baselines for Semantic Code Search**. In: *Proceedings of the 1st Workshop on Natural Language Processing for Programming (NLP4Prog 2021)*. Aug. 2021 (see pages 43, 45).

Declaration of Authorship

I hereby declare that this thesis is my own unaided work. All direct or indirect sources used are acknowledged as references.

Potsdam, September 12, 2024

[illegible]

Till Wenke

A Feature details

Top level directory feature

I want to demonstrate that issue descriptions indeed seem to contain some information that helps to infer the top level directory in which files lie that have to be changed to solve the issue. Proofing this, can already be an indicator whether this feature can be beneficial for the retrieval task. For this purpose, I show that given an issue description and a directory a BLPR model can partially predict whether a change has to be made in this directory.

Specifically, I build a dedicated dataset from the Kafka retrieval dataset with only the top level directory feature where one sample contains an issue i and a top level directory t. The target values become 1 if any file in t has to be changed to solve i. Eventually, duplicate samples are removed and the dataset is split according to a 70/30 train/test-split. For the resulting dataset 28% of the samples are positive.

The model is initialized with the prior mean of 0 for all weights, $f_d = 5$ and the prior for the bias term is chosen as $\mathcal{N}(-0.9, 1.0^2)$ (see Section 4.2.3).

Evaluating the trained model on the test set, I yield the ROC curve in Figure A.1 which demonstrates that the model is to some extent able to decide whether a directory will contain a file change with respect to an issue. Consequently, it is worth considering the top level directory further for the file retrieval task.

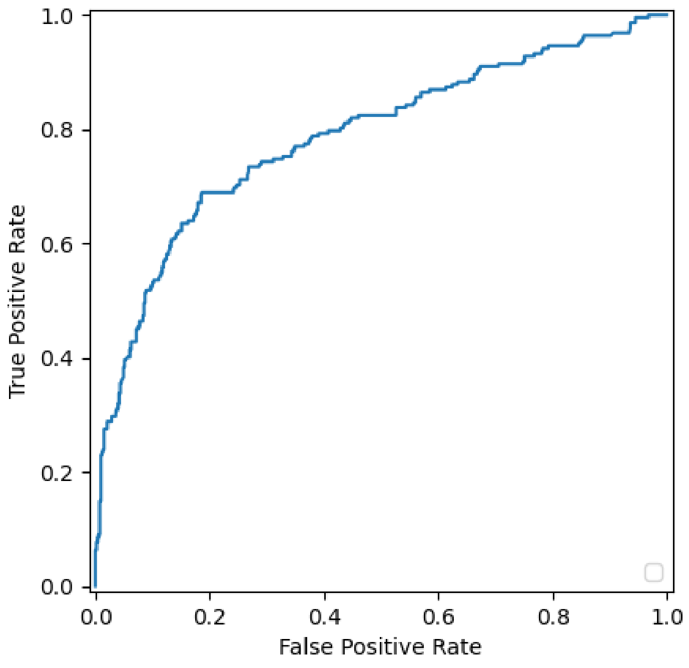


Figure A.1: ROC curve for a BLPR model predicting changes in directories from issues for the *provectus/kafka-ui* repository.

B Discussion

Shift in ranks by binary features

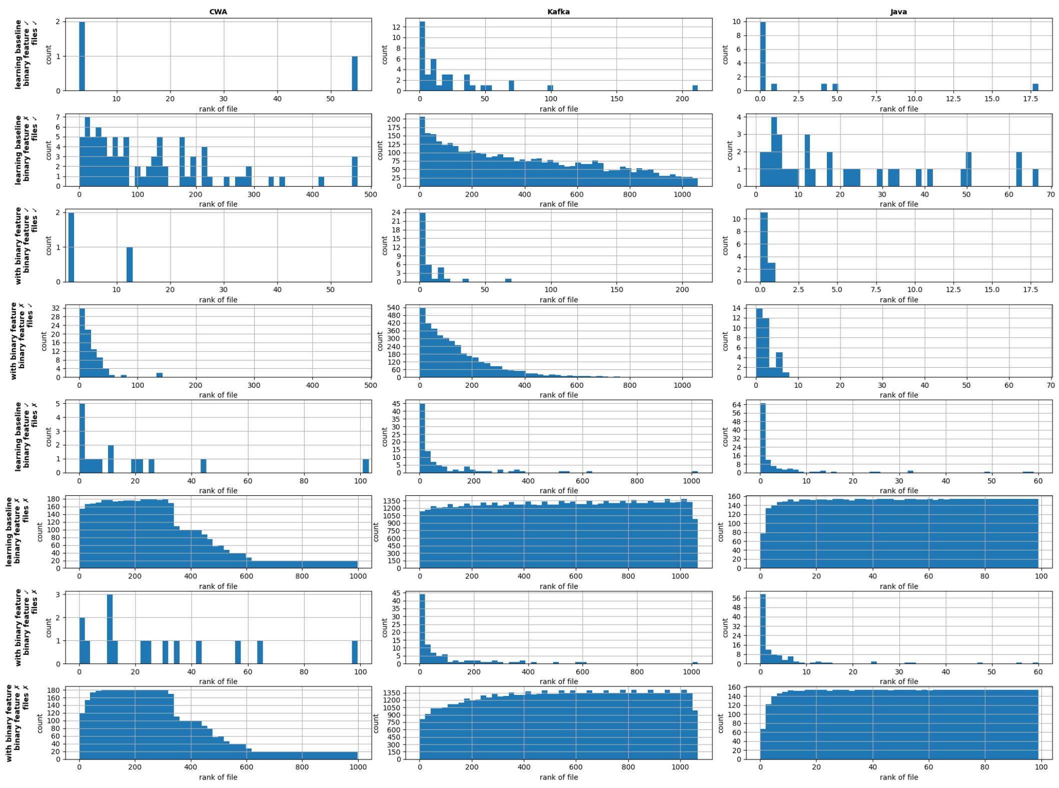


Figure B.1: Impact of introducing the *author feature* on the ranking of files for every retrieval dataset, displayed separately for (in)active binary feature and (ir)relevant files.

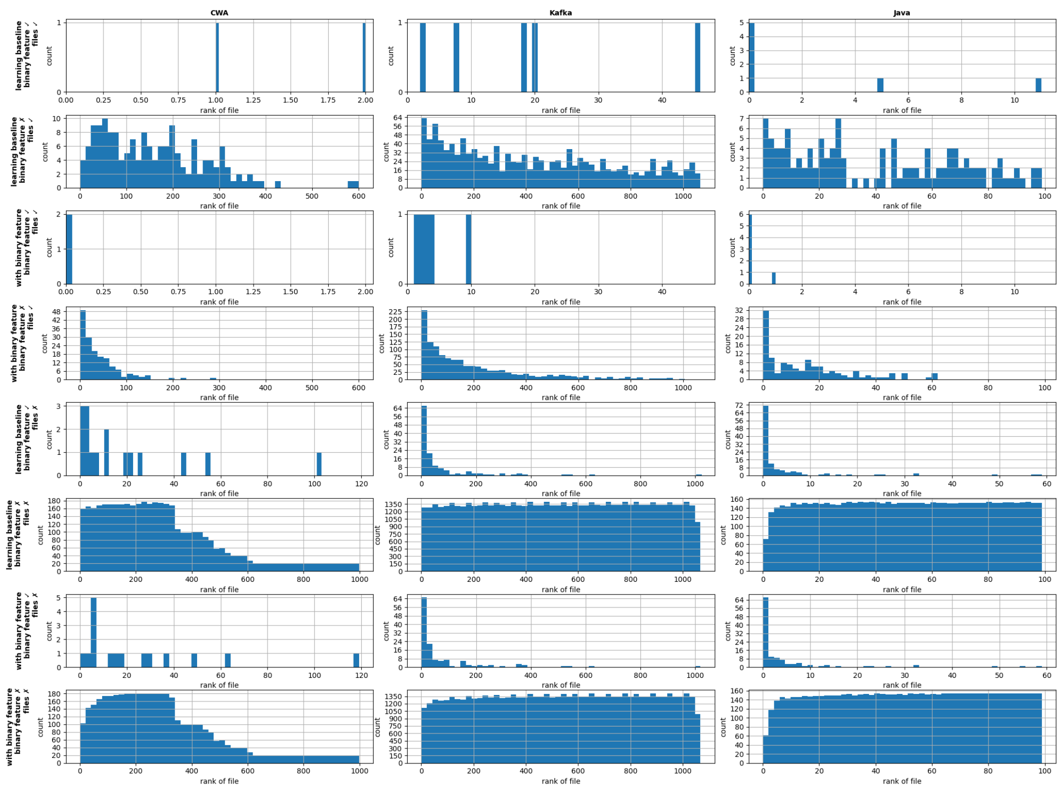


Figure B.2: Impact of introducing the *binary recent file change feature* on the ranking of files for every retrieval dataset, displayed separately for (in)active binary feature and (ir)relevant files.