MASTER'S THESIS 2023

Recommending Relevant Open Source Software using Semantic Functionality Search

Filip Hedén, Nils Barr Zeilon

ISSN 1650-2884
LU-CS-EX: 2023-08

DEPARTMENT OF COMPUTER SCIENCE
LTH | LUND UNIVERSITY

EXAMENSARBETE
Datavetenskap

LU-CS-EX: 2023-08

Recommending Relevant Open Source Software using Semantic Functionality Search

Rekommendation av relevant öppen programvara med hjälp av semantisk funktionalitetssökning

Filip Hedén, Nils Barr Zeilon

Recommending Relevant Open Source Software using Semantic Functionality Search

(Using clustering of function embeddings to recommend relevant open source software repositories)

Filip Hedén
fi6468he-s@student.lu.se

Nils Barr Zeilon
his15nze@student.lu.se

March 29, 2023

Master's thesis work carried out at Debricked AB.

Supervisors: Christoph Reichenbach, christoph.reichenbach@cs.lth.se
Emil Wåreus, emil.wareus@debricked.com

Examiner: Niklas Fors, niklas.fors@cs.lth.se

Abstract

In modern software development engineers are not expected to write software in isolation, but through open sourced software components and libraries reuse and refine work that has come before. With the vast quantity of open source libraries it is more important than ever to easily be able to find the right library for a specific need. This thesis introduces the idea of creating repository embeddings, representing functionality contained within an open source library, through clustering of embeddings created from source code functions found in open source software repositories.

By embedding a natural language query to the vector space of the repository embeddings, we are able to recommend software repositories based on multiple functionalities provided by the repository in an efficient way.

To evaluate the approach we conducted a user study to gather annotated data on the relevance of open source repositories for given search queries. The user study resulted in almost 50 annotated queries and 500 annotated repositories. When this annotated dataset was used to evaluate the recommendation models it was found that using function embeddings directly was in general the best performing approach, but that clustering embeddings were not far behind. It was also found that the best approach was to combine function and cluster embeddings.

We conclude that using function embeddings can help in recommending relevant software repositories, and further that using clustering can improve these recommendations. We also found that using clustering embeddings can be a way of increasing the scalability of functionality search compared to using function embeddings. Finally we found that separation of clusters correlates with relevance of recommendations when using clustering algorithms for the task of functionality search.

Overall the results indicate that the idea has potential for both increased relevance and scalability, but more data is needed and there is much potential for improvements in future works that could make this system viable for real world usage.

Keywords: Semantic Functionality Search, Clustering, Recommendation Systems

Acknowledgements

Firstly, we want to thank Emil Wåreus, our supervisor at Debricked who came up with the idea for the thesis as well as helping us throughout all parts of the project, without whom this thesis would certainly not exist.

We also want to thank Christoph Reichenbach who provided exceptionally detailed feedback on this thesis and ensured the thesis would even be presentable.

Further we want to thank Anton Duppils and Magnus Tullberg at Debricked for their invaluable help in many discussions that lead to this thesis and helping us with all technical issues and setup along the way.

We would also like to extend thanks to Niklas Fors who graciously agreed to examine this thesis.

Finally we would like to thank our friends at LTH and our coworkers at Debricked who participated in the survey and made the thesis possible. Thank you.

Contents

Chapter 1
Introduction

1.1 Motivation

As software engineers we are expected not just to write software, but to reuse and extend other developers' software. Open source software is often distributed in the form of a software repository, a collection of source code and metadata files that describes the software component.

There are many good reasons to reuse code published as open source; it saves time when building software, improves quality and reduces work needed for maintenance [13]. In addition to this, as the quantity and quality of open source software increases, the probability that a good quality open source software component exists that suits a problem being solved by a software engineer increases.

1.2 Problem statement

The software engineer has a problem however, how does she find this open source software component? It is available to her, residing in a software repository available freely on the internet. But so are a million other repositories, making it hard to find the one that solves her problem, i.e. is relevant to this specific developer. This is what we will refer to as the problem of **relevant software discoverability**.

So far regular search engines, a type of information retrieval system, have been used to find this relevant software. And while they are helpful they usually only take into account the metadata of software repositories, such as author added labels or short descriptions, which doesn't necessarily reflect the actual source code contents of the software repository. As the description can either be false or completely missing. Regular search over source files might work, but the software engineer in search of a solution to a specific problem may well indeed not have a single clue what the code would look like in the first place.

Currently solutions exist that can match and return functions from source files, by describing the desired functionality of the function. But such solutions are not taking into account the higher abstraction of open source repositories, which is the most common way of distribution today. Also, returning only a function instead of a ready to use repository may encourage users to copy-paste results, missing one great feature of using open source software, namely that you do not need to maintain all of the code yourself.

Which begs the question if it is possible to produce a system which not only can return specific relevant functions but also taking into account the utility of the repository in which the function resides. Returning relevant code functionality at a repository level to the end user.

Indeed, researchers have been trying to create such a system for at least two decades [21].

1.3 Approach

What we envision for the future of open source search is a system that can return to its user software repositories dedicated to some functionality relevant to solve a specific problem. The system we envision would be able to take as input a description in English, or any other natural language, and based on the functionality, or semantic intent, of the description return to its user multiple alternatives of matching software repositories [20]. Using a natural language query to describe the functionality is a natural extension to what users today are already comfortable with, from the use of more general search systems like Google and Bing.

Firstly, a system of this kind would need to be able to capture the semantic intent behind text written in a natural language. Fortunately, there have been many advances in machine learning models for natural language processing, the research area concerned with making natural language machine understandable, over the last decade that enable powerful semantic parsing of natural language. One of these advances are large language models, statistical models consisting of many layers of neurons which are able to quite accurately interpret the semantics of natural human language, capturing more complex parts of the language such as semantic intent [10].

Secondly, proper recommendation of source code for a given natural language query requires that we can represent the functionality described by the source code as well. We are fortunate that the last decade's advances in language models have proven successful for this problem as well. Today's large language model can not only be trained to interpret the semantics of written human natural language, but also written source code in programming languages.

So far, what we have described is what is required by systems that are already in place today, these systems are able to match between representations in natural language and programming language. Making it possible to describe a desired functionality and recommend a code-snippet, a short piece of code in the form of a function, that solves the given task. However, as mentioned before, our software engineer might be more interested in finding a software repository that contains the code-snippet or function as before but in addition a whole set of functions that work together to handle more complex tasks, referred to as a library or software component. This task is the aim of this thesis. Since we are looking at components or libraries to reuse, not singular functions, we want to be able to search for a more abstract level of functionality or semantic intent. A component or library usually con-

sists of a lot of source code, covering many different semantic intentions. One example would be the popular Python package "pandas", a library for data analysis [15]. While this library of course includes algorithms for statistical analysis, it also has functionality for reading and writing to various file formats and interacting with memory efficiently.

Thirdly, a system such as the one we envision would require an efficient way of representing all of the functionality contained in an open source software repository. To illustrate the problem let's look at the example above once more. "pandas" contains thousands of functions in its library, each one contributing to the component as a whole. While pandas contain many functions concerned with outputting things in a nice format, it would not be called a formatting component, as those functionalities are only there to enable other parts of its system, let's say. So individual functions might not be the best way to represent a library. Our idea is that while individual functionalities might not say very much about a library's intended use case, grouping of such functionalities might. If we could group together functionalities within the component that are similar to each other the intent of the component might emerge more accurately and efficiently. Grouping together things in such a way is referred to as clustering. To bring the idea back to our example of "pandas", the clusters that emerge could hypothetically represent something like "statistical analysis", "file handling", "memory optimization" which are more closely related to what the library declares in its description.

In this thesis we will research if we are able to utilize clustering to categorize groups of functions with similar semantic intent inside of open source software repositories to then recommend relevant repositories using natural language search queries.

A search engine that provides relevant recommendations for software components has the potential to greatly help in code reuse, because it greatly simplifies the process of finding the right software to reuse, i.e. increasing relevant software discoverability. This has many benefits for the industry. The most obvious benefit is that less time needs to be used for writing source code, but also that less time is needed to evaluate source code that can potentially be reused. Another benefit of reusing software components is that less time is needed for maintenance. Finally, less source code and software means that it is easier to ensure security. In addition to reducing the time and effort spent by security analysts and cost of security vulnerabilities being used maliciously, more reliable open source software can help adoption which further reduces code duplication in the industry. Overall, such a search engine could have a great positive impact on the industry as a whole.

1.4 Research Questions

To measure the effectiveness of our approach we have set up a number of research questions. However before presenting them some brief clarifications are in order. Our thesis is concerned with embeddings, a form of numerical representation of the semantic intent of natural language or the functionality of a code function. It is also concerned with clustering, which is another word for grouping together similar objects. In this thesis our aim is to cluster, or group together, similar function embeddings.

- **RQ1**: To what degree can we utilize embeddings to find functionality in software repositories?

- **RQ2**: To what extent can aggregations of embeddings improve the quality of search results of functionalities within software repositories?
- **RQ3**: To what degree can aggregations of embeddings increase scalability of functionality search of software repositories?

  When aggregating information, which in this thesis is done by clustering the data, many different forms of clusters can emerge. At which point it is relevant to look at the quality of different clustering techniques. Separation is one such way of determining the quality of resulting clusters.
- **RQ4**: To what extent does higher quality separation of clusters of functionality within a software repository relate to relevant functionality search results using our approach?

1.5 Scope

We restrict our investigation to code snippets from one programming language, a single language model for producing embeddings, a set of three aggregation methods and a single dataset for repositories and functions. The choices are explained further in the following sections.

1.5.1 Programming Language

In the study we limit ourselves to one programming language, Python. This is both due to time constraints and because we need to test our search engine on subjects familiar with the language. Due to the latter of these Python fits well because developers at Debricked and many students at LTH use Python and have knowledge about its ecosystem.

1.5.2 Language Model

We limit ourselves to a pre-trained language model, UniXCoder [9], that we do not train further for our specific task due to time constraints. This is partly because we do not believe we would be able to improve upon the state of the art in embedding creation over the course of a thesis project. It is also partly due to our focus being on utilising the language model for a specific task, rather than improving a language model's performance on a task it's already tested for.

1.5.3 Clustering Algorithms

We limit ourselves to a set of two popular unsupervised clustering algorithms. K-Means clustering and Ward clustering. There exists a vast number of clustering algorithms, but due to the scope of this thesis testing them all would not be possible.

In addition to this we compare with grouping together functions based on their position in the repository file tree.

1.5.4 Dataset

We limit the data used for testing the model to the data available from CodeSearchNet [11]. The reason for this is that the data is well analyzed by multiple papers, already clean enough to not require further parsing, and contains a large amount of useful Python open source projects hosted on one of the most popular version control system hosting platform today; GitHub, hosting over 330 million software repositories [1]. We believe that this makes the dataset quite representative of Python open source projects in general.

Chapter 2
Related Work

In this chapter we want to give a tribute to the works that have made this thesis possible.

First, the paper "CodeSearchNet Challenge Evaluating the State of Semantic Code Search" [11] has been instrumental to this thesis as it provided insight and inspiration into the field of code related retrieval tasks. It also provided a large dataset, which without this thesis would possibly have taken a much different turn.

Second, the paper "Sosed: a tool for finding similar software projects" [7] acts as a forebearer to what this thesis tries to accomplish. It presents search among software repositories in much the same manner as this thesis, with the use of code embeddings.

2.1 CodeSearchNet

The paper "CodeSearchNet Challenge Evaluating the State of Semantic Code Search" [11], was an effort to further research in the area of Semantic Code Search. The research conducted by this project included the task of "retrieving relevant code given a natural language query", which is closely related to what this thesis objectives.

In the paper the authors presented a challenge to achieve an as high score as possible for Normalized Discounted Cumulative Gain [12], a popular metric for recommendation systems, for a set of natural language queries and source code snippets.

The authors also released a large dataset of doc-string and source code pairs to train models for the task.

2.1.1 Similarities

What makes this paper similar to our thesis is its goals and procedures.

We both share the same goal of finding better systems of high quality code retrieval. They focus on code-snippet retrieval while we focus on software repository retrieval. The user interface to the user of such a system is also very much alike, querying a dataset with a

natural language query and returning matching code with the described functionality from the query.

We also utilize the very same data set that was presented by the CodeSearchNet challenge in our research and use the same search metric to evaluate our model.

2.1.2 Differences

The main difference to this thesis is the abstraction level of the code retrieval task. Our investigation changes the task retrieval of code-snippets to software repositories, and how many such code-snippet embeddings can work together to form an embedded representation of software repositories. One can think of it as going up one abstraction level of retrieval of code functionality.

2.1.3 Results

CodeSearchNet was a challenge and thus provided results both in the initial paper and on a leaderboard. The best normalized discounted cumulative gain score presented in the paper was 0.45 for Python, and the best score for Python on the leaderboard was 0.47 [2].

2.2 Sosed

The paper "Sosed: a tool for finding similar software projects" [7], from JetBrains research studied how clustering embeddings could help find similar software repositories.

The paper introduces a model which takes as input a software repository, extracts tokens of the source code if the programming language is supported, creates embeddings and computes cluster distribution to then match the repository to the one with the closest cluster distribution in the search dataset.

2.2.1 Similarities

The project is similar to ours in that it utilizes source code embedding to categorize parts of software repositories. It also recommends other software repositories based on matching embeddings from the input.

2.2.2 Differences

Our main differentiator from "Sosed" is that instead of taking a software repository as input for similar repository recommendations, we want to take as input a query in natural language and recommend relevant repositories. While we use a similar approach to categorize software repositories and what they semantically do, our end goal is quite different.

Chapter 3
Background

In this chapter we will introduce the various concepts that the thesis builds upon.

As a starting point we will introduce the reader to the field of Linguistics, the study of human languages. As the thesis is concerned with the analysis and retrieval of textual information in human languages, a good understanding of human languages is a cornerstone to understanding how all other parts of the thesis are connected.

In the second section we will give an overview of the field Natural Language Processing, the art of machine processing of textual information. We start off by introducing the concept of an *embedding*. The questions that we set up for this thesis are all revolving around functionality embeddings, and therefore a thorough understanding of what exactly function embeddings are is essential for the understanding of this thesis. We then follow by introducing the various techniques that allow us to create embeddings from both code and English text.

In the third section of this chapter we introduce the reader to Information Retrieval, as this thesis is concerned with creating a system for retrieval of functionality within software repositories the reader must understand what exactly constitutes a good information retrieval system.

In the final section we introduce the concept of *clustering*, along with the clustering techniques which were used in our work and how they work. As many of our initial questions to this research project were concerned with if aggregations of function embeddings could produce better information retrieval systems, and to that extent this section is vital to how these aggregations are made.

3.1 Linguistics

Linguistics is the study of human languages.

Linguistics has historically been divided into levels, all building on each other to create meaning. The first level of a language is **phonetics**, or sounds. The second level is **mor-**

phology, how sounds are combined into words. The third is **syntax**, how words are formed together to create sentences. The fourth is **semantics**, how meaning is derived from the words and sentences. The fifth and final level is **pragmatics**, how meaning is derived not only from the sentence itself but also from its context [17].

In our thesis the branch of semantics is of special interest, although strictly speaking it builds on the lower levels of linguistics in the context of human speech, we are more concerned with methods of finding the meaning of sentences than correctly representing phonetics, morphology and syntax within sentences.

3.1.1 Semantics & Meaning

Semantics is concerned with deriving "meaning" out of words and sentences. "Meaning" in this context should be interpreted as a form of conceptualization or reference, conveying ideas through language as well as relations between ideas. In the English language for example, the word "dog" is a reference to a type of animal.

In the task laid out by this thesis, "meaning" refers to the functionality of a code snippet, i.e. what task a certain piece of code can accomplish.

Semantics can be very hard to capture, much harder than syntax in a language. One reason is because there exist sentences that may be syntactically correct, but hold no meaningful information or the meaning has been lost.

From the perspective of a programmer, this is perhaps more easily understood as writing programs that are syntactically correct but produce no meaningful results. For example, the sentence *Colorless green ideas sleep furiously* [8] is syntactically correct, yet it holds no meaning to us.

3.1.2 Ambiguity

Understanding human languages might seem like a simple problem, but it is easily forgotten that the learning process often takes decades to develop fully for most humans, so to teach a computer to understand human languages may feel like a daunting task. One reason for this is ambiguity, where a word or sentence has multiple meanings. The word "bank", for example, may mean both an economic institution and the side of a river [6].

This is also true for sentences, such as the sentence *We saw her duck* [17]. Depending on its context, the word "duck" could mean either the bird belonging to a woman or the motion of the woman avoiding something.

3.1.3 Programming languages

Programming languages can be much different from naturally spoken languages. Programming languages are formal languages, there is a notion of correctness of the language. Programming languages are usually subject to much stricter constraints than naturally spoken languages, this is to reduce ambiguity in the language to make it easier to automatically process by computers. The constraints that are put on the language when writing code can be well handled by a computer program called the compiler, which translates between the programming language and byte-code that can be run on the system.

Much work has been done in the processing of programming languages this way, and we can build on a large body of work when analyzing programming languages. However we are interested in conveying meaning in the form of a natural language query and receiving matching snippets of programming languages. How can we go from something that is ambiguous to something that is much stricter?

3.2 Natural Language Processing

Natural language processing or NLP, a sub-field of both linguistics and computer science, aims to create models of human languages to make them machine understandable, making it possible for machines to understand textual information, i.e. map words and sentences to ideas as well as relations between ideas [17].

3.2.1 Embeddings

A common approach in Natural Language Processing is to represent textual information as numerical vectors, which allows mathematical transformation of the textual data. There are many types of such vectors that we will refer to as embeddings. For example there exists both character and word embeddings, with different aims in their scope of capturing relationships within the textual data.

The goal of a word embedding for example, could be to represent words that have closely related meaning closer together in the vector space. One property of numerical vectors is that they are additive, combining two vectors will yield another vector with a different meaning. For example, taking the vector representation of Paris, the capital city of France, minus the vector representation of France plus the vector representation of Poland might yield the vector representation of Warzaw, the capital city of Poland. In this example, the information or meaning contained in the vector Paris-France, would be something like “Capital City”. After adding the vector representation of Poland what we end up with is a vector representation containing the information of “Capital city of Poland” i.e. “Warzaw” [16].

In regard to this thesis, the embeddings that we are interested in are such that they capture important syntactic and semantic relationships between words or sentences[16]. To answer the questions posed by this thesis, we are interested in mapping the textual information contained in a code-snippet to an embedding which we refer to as a “function embedding”. Ideally the embedding of such a code-snippet represents the functionality provided by the code.

Word embeddings by themselves will still contain all the ambiguity of the language they are built from, for instance the word “bank” from our previous example can still hold multiple meanings. To handle the problem of ambiguity a common approach is to gather additional information from the word surroundings in the embedding stage. As vectors are additive, we can add information from the word embeddings surrounding the word to gain more clarity in its meaning. For example, if we wish to embed the word “bank” in the sentence “he went to the bank, to collect his monthly salary”, we could take the embedding from the word bank and add weighted vectors of the contextual words “he”, “went”, “to”, “the”, “to”, “collect”, “his”, “monthly”, “salary” to the embedding to give it some context, nudging the vector in the direction of a hypothetical embedding describing “financial institution”.

This approach to embed words from their context can be done statically, taking a non-changing window size to embed words by their immediate surroundings.

3.2.2 Cosine Similarity

Representing textual information in the form of embeddings allows us to build upon a large corpora of research in the field of linear algebra. In the context of this thesis, we are for example interested in analyzing how closely connected two different functions are in what functionality they contain. The idea being that if functionality is mapped to different parts of some vector space, one could map embeddings of English language to the same vector space and retrieve the closest function embeddings to the English embedding and hopefully retrieve functionality that match its English description.

Ideally if we compared three embeddings, two of which describe the same functionality but in different styles of programming and the third some completely different functionality, the two embeddings describing the same functionality would be closer together in vector space compared to any pair including the third embedding. But how can we measure such a similarity between embeddings?

A common technique in Natural Language Processing is using "Cosine similarity" which is one way of comparing similarity between vectors by taking the cosine value of the angle between the two vectors [14]. There are many other techniques to compare vectors, such as considering the magnitude of the vectors compared and not just the angle between them. However following the UniXCoder [9], which in an example of fine-tuning the model for code search utilize the "cosine similarity" for comparison between embeddings, we decided on limiting our research to the use of "cosine similarity" for determining the distance between a pair of vectors.

The cosine value can be found using the dot product between two vectors using the following formula:

$$\cos(\theta) = \frac{\mathbf{A} \cdot \mathbf{B}}{||\mathbf{A}||||\mathbf{B}||} = \frac{\sum_{i=1}^{n} A_i B_i}{\sqrt{\sum_{i=1}^{n} A_i^2} \sqrt{\sum_{i=1}^{n} B_i^2}}$$

3.2.3 Abstract syntax tree

An abstract syntax tree (AST) in computer science is an abstract representation of the syntax in a piece of text from a formal language such as programming languages. It is mainly used in the compilation stage as a way to analyze the code structure to make sure that it adheres to the structural requirements set by the compiler.

It is said to be abstract as not all details of the syntax is part of it, instead, each node in the tree represents a construct, often a data type or an operation requiring a particular set of data types, of the programming language.

3.2.4 UniXCoder

UniXCoder is a pre-trained multi-modal language model for programming language and natural language [9].

Pre-trained models in the field of machine learning, is referring to when a model has already been trained on a large dataset as to give a starting point before directing the model to a specific task, removing the need to retrain a model for specific tasks. In Natural Language Processing it is common to use what we refer to as language models, usually giant neural network architectures which are trained on large datasets of textual data where its task is to predict continuation of a sentence.

Multi-modal in this context means that the model takes its inputs from multiple input sources, in the case of UniXCoder it takes a pair of textual data in the form of both a comment and a flattened abstract syntax tree of the code function.

UniXCoder is based on a specific such neural network architecture referred to as a transformer, introduced in the canonical paper "Attention is all you need" [22]. The architecture allows for efficient encoding of word embeddings from training. The architecture takes a sequence of textual embeddings, commonly referred to as tokens, and is trained in such a way as to learn the pairwise importance of relations between tokens in the sequence. This allows for a better encoding of tokens, as the context of each token can be embedded in an efficient way.

To make the Abstract Syntax Tree (AST) representation of the code compatible with the Transformer architecture UniXCoder introduces a one-to-one mapping function that takes an Abstract Syntax Tree and transforms it to a sequence, while still keeping all of the structural information intact. Pseudo-code for the mapping function is provided in figure 3.1. The function, f, works recursively, starting with the root of the tree. If a node has no children the name of the AST-node is returned. Whenever an AST-node has children, f surrounds the child with the tags containing the name of the node

$$< name, left > f(child_i) f(child_j) ... < name, right >$$

.

UniXCoder is trained to be able to operate in three different modes, encoder-only, decoder-only and encoder-decoder. For the purposes of this thesis we are only interested in the encoder option, which allows us to use UniXCoder as a mapping function from a piece of textual information, a comment or a code-snippet, into an embedding in vector space.

Encoding-only mode is achieved by training the UniXCoder in a Masked Language Modeling setting, where a small set of tokens in a sequence are masked i.e. replaced by random tokens, where the training goal of the model is to predict what the original tokens were before masking. This allows the model to efficiently infer the masked tokens based on the semantic and syntactic information in the comment-AST pair. With the encoding mode enabled UniXCoder outputs an embedding, from the context of an entire code function or natural language string, a numerical vector of size 768. An overview of the model architecture of UniXCoder in encoder-only mode is provided in figure 3.2.

Algorithm 1 AST Mapping Function $\mathcal{F}$

Input: The root node $root$ of AST
Output: A flattened token sequence

```
function F(root)
    seq = an empty list
    name = the name of root
    if root is a leaf then
        seq.append(name)
    else
        seq.append(name :: left)
        for child in children of root do
            seq.extend(F(child))
        end for
        seq.append(name :: right)
    end if
end function
```

Figure 3.1: pseudo-code for the AST to flattened-AST transformation. Image taken from UniXcoder: Unified Cross-Modal Pre-training for Code Representation [9]

3.3 Clustering

Clustering is a form of grouping together of data points in such a way that those belonging to a group, referred to as a cluster, are more similar to each other than to those in other groups. Clustering can either be done using either a top-down or bottom-up approach. Top-down in clustering is where you begin with randomly selecting which cluster a number of points belong to, and over time using an algorithm fitting more accurate clusters by displacing points from one cluster to the other. A bottom-up approach means that each point starts out as its own cluster, and over time clusters are merged based on their position relative to one another.

Clustering is often done in an unsupervised or semi-supervised manner. Unsupervised means that there are no previously human-labeled data points. With respect to clustering, unsupervised clustering means that the methods used for clustering build only from the inherent structure of the data without any supervision from human input. In other words there is no ground truth to which clusters should emerge before seeing how the data is structured [14]. Semi-supervised with respect to clustering is when some of the features of the clusters is determined beforehand, in many algorithms it is common to determine the number of clusters that should emerge for example.

Many of the questions we posed for this thesis relate to aggregations of function embeddings, and clustering is one such method to aggregate information. In the context of this thesis, the way we aggregate embeddings from a cluster or group is by taking the mean value of all embeddings belonging to the cluster.

In the following sections we will present the techniques that we used in this thesis in clustering of function embeddings. We present two common techniques with k-means, which is a semi-supervised top-down algorithm, and ward which is another semi-supervised but bottom-up algorithm. In this thesis we chose the two to be able to compare the results be-

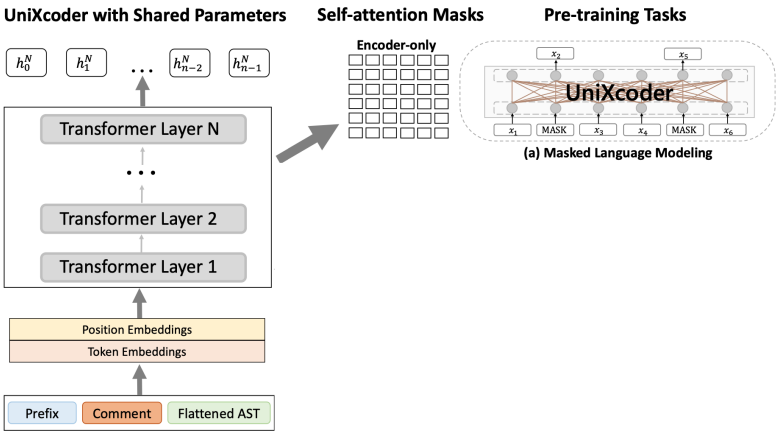

Figure 3.2: UniXCoder architecture and encoder-only mode. Image taken from UniXcoder: Unified Cross-Modal Pre-training for Code Representation [9]

tween a top-down and a bottom-up approach.

3.3.1 k-means

K-means clustering is a technique for dividing a vector-space of n embeddings into k clusters.

The method is initialized with k initial cluster-centers, then the method iterates the following steps until it either converges or is stopped. Convergence is met when none of the assigned vectors change cluster k from one iteration to another [14].

- **step zero:** k randomly selected points in the vector space are chosen as the starting clusters. These points will be referred to as cluster centroids.

iterate until convergence:

- **step one:** cluster assignment each embedding in the vector space is assigned to it's closest cluster based on a distance metric, where we used the euclidean distance.

$$S_i^{(t)} = \left\{ x_p : \left\| x_p - m_i^{(t)} \right\|^2 \leq \left\| x_p - m_j^{(t)} \right\|^2 \ \forall j, 1 \leq j \leq k \right\},$$

Here, t denotes the iteration number. $S^{(t)}$ denotes the set of k clusters at iteration t. x_p denotes a single data point. m_i m_j denotes two different clusters centroids.

- **step two:** recalculation of cluster centroids each cluster centroid is recalculated to be the mean of the embeddings assigned to it.

$$m_i^{(t+1)} = \frac{1}{\left| S_i^{(t)} \right|} \sum_{x_j \in S_i^{(t)}} x_j$$

Here, m_i^{t+1} denotes the cluster centroid of cluster i in the next iteration $t+1$. $S_i^{(t)}$ denotes the cluster i at the current iteration step t.

3.3.2 Ward

Ward's clustering algorithm is an agglomerative hierarchical clustering algorithm, that is a bottom up approach to finding clusters in the data, each observation starts out a a singleton cluster and after each iteration it merges the two clusters that have the smallest increase in the total within-cluster variance, also called inertia, until there exists k clusters, specified by the user [23].

In each iteration, two clusters are merged under the constraint

$$d_{i,j} = d(\{X_i\}, \{X_j\}) = \|X_i - X_j\|^2.$$

That is, the clusters X_i, X_j are merged into X_{ij} if X_i, X_j minimize the sum of the squared difference between all pairwise clusters.

3.4 Clustering Metrics

To compare different clustering methods and to evaluate how good a clustering is we need metrics. A commonly used metric for evaluating clustering is the Silhouette Score, which is described in detail below.

3.4.1 Silhouette Score

The Silhouette Score is a measure of how well-separated the clusters in a clustering algorithm are. It was introduced in the paper "Silhouettes: a Graphical Aid to the Interpretation and Validation of Cluster Analysis" [18] and is a measure of both tightness and separation of a cluster in comparison to all other data points.

It ranges from -1 to 1, with a value of 1 indicating that the samples in a cluster are completely separate from the samples in other clusters, and a value of -1 indicating that the samples in a cluster are more similar to the samples in other clusters than they are to the samples in their own cluster. A value of 0 indicates that the samples in a cluster are not well-separated from the samples in other clusters [18].

For a point i in a cluster C_I the mean distance between i and all other points in the cluster is calculated as follows:

$$a(i) = \frac{1}{|C_I| - 1} \sum_{i \in C_I, i \neq j} d(i, j) \tag{3.1}$$

Here $d(i, j)$ denotes the distance between a point i and a point j both belonging to a cluster.

This can be interpreted as how well the point fits in its assigned cluster.

The mean dissimilarity between a point i and some cluster C_J as the distance between the point and all points in C_J where $J \neq I$, and to find the least dissimilar cluster to the point i using the min operator:

$$b(i) = \min_{J \neq I} \frac{1}{C_J} \sum_{j \in C_J} d(i, j) \tag{3.2}$$

The cluster with the least mean dissimilarity for a point i is said to be its neighboring cluster.

Finally, the silhouette score is defined as

$$s(i) = \frac{b(i) - a(i)}{max\{a(i), b(i)\}}, \text{ if } |C_I| > 1 \tag{3.3}$$

and

$$s(i) = 0, \text{ if } |C_I| = 1 \tag{3.4}$$

Which will always be a value between -1 and 1, which can be interpreted as how well the given point fits in its cluster.

To determine how good the clustering of a dataset is we can calculate the mean silhouette score for points in the dataset.

3.5 Information Retrieval

Information retrieval, or IR, is the scientific field of search for information. This could be searching in documents, databases, videos or any other source of information.

An information retrieval system is a software system that provides access to whatever data or information a user is searching for.

The process of information retrieval starts with a query into the information retrieval system. A query is a statement of information needs, for example a string in a search engine like Google (https:www.google.com). An important thing to note is that a query does not correspond to a unique object in the information source, but may instead match many different objects to varying degrees of relevance.

Information retrieval systems usually compute some kind of score of how well each object in the source matches the query, which leads to a ranking of the information objects. The objects with the highest score are presented to the user [14].

One common way to rank to match information in a collection to a given query is to use a vector space model. This model will be discussed further in the next subsection.

3.5.1 Vector Space Search

Vector space search is a sub field of information retrieval for textual information. Using the vector space model, each object in the collection is represented by a vector. The vector space model was first introduced in the SMART document retrieval system [19].

To match a query to the best match in the collection, it is first vectorized with the same model used to vectorize the objects in the collection. The distance between the query vector and the object vectors can then be calculated to find the most similar vectors and thus the best matches. In practice, the angle between vectors using cosine similarity is often used to measure similarity between vectors [14].

3.5.2 Recommendation Metric

To evaluate how well an information retrieval system works, some metric for determining how relevant the recommended search items actually are is needed. There exist many metrics for evaluating search relevance, but we elected to use the same metric as CodeSearchNet, Normalized Discounted Cumulative Gain, as much of our work is built on that paper [11].

Normalized Discounted Cumulative Gain

Normalized discounted cumulative gain (NDCG) is a common metric for evaluating information retrieval systems [14]. The metric uses relevance of retrieved items and its rank to give the recommendations for a query a score.

The first part of calculating the NDCG is to calculate the discounted gain of a recommended item at a given rank:

$$DG(r) = \frac{R(r)}{log_2(r+1)} \tag{3.5}$$

where r is the rank of the recommended item and $R(r)$ is the relevance of the item at rank r.

The "discount" part of discounted gain makes a relevant item appearing at a lower rank, or earlier in the search results, more valuable than an equally relevant item appearing at a higher rank.

It is also common to transform the score to further emphasize the difference between high and low relevance items using the following formulae:

$$DG_{emph}(r) = \frac{2^{R(r)} - 1}{log_2(r+1)} \tag{3.6}$$

For the "cumulative" part of NDCG we just sum the discounted gain values for all items recommended by the model:

$$DCG = \sum_{r=1}^{k} DG(r) \tag{3.7}$$

where k is the number of recommended items for the query.

A problem with the discounted cumulative gain for measuring recommendations is that the score for one query can be much higher than one for another simply because of the annotations for the different queries. To remove this fallacy NDCG introduces normalization by dividing the computed DCG score by the ideal DCG score for the given query:

$$NDCG = \frac{DCG}{IDCG} \tag{3.8}$$

where IDCG is the ideal discounted cumulative gain.

IDCG is calculated in the same way as DCG but instead of using recommendations from a evaluated model the recommendations are replaced by sorting the dataset on annotated relevance to achieve maximum possible DCG from the annotated dataset.

NDCG can be a useful metric of recommendations, but does have a problem in that poor recommendations can result in a high score if no good recommendations are annotated for the dataset used in evaluation, i.e. if all annotated items are of the lowest possible score then any possible combination of recommendation would achieve maximum NDCG. Another problem with the metric is that an evaluated model can return relevant items that are not annotated, which would result in a bad score for a result that is actually good.

Chapter 4
Approach

In this chapter we will present each step that was necessary for us to answer the research questions laid for this thesis. As this thesis is concerned with properties of information retrieval systems, a minimal such system had to be constructed and tested. An outline of this system's architecture will be described in this chapter.

In the first section we present the dataset that we used in the thesis. To validate the results of our recommendation system or search engine, we needed a dataset containing code functions as well as metadata with attributes such as which open source repository it belonged to and a matching url for further inspection. Especially important to this thesis was that the dataset was of sufficient size and diversity in functionality.

In the second section we present the inner workings of the recommendation system. How embeddings and clusters were produced and how embeddings were compared in relation to a natural language query to the system to give a ranked list of recommendations to the user of the recommendation system.

In the third section we present our approach of evaluating a recommendation system or search engine, using a user study. We present how the user study was set up, how we chose the participants and a detailed explanation of the survey. The conduction of the user study was vital to answer all of our research questions, and indeed without it this thesis would not have been made possible.

We end this chapter with how we conducted the evaluation of our recommendation system. Evaluating both the results from the user study as well as statistical properties related to our dataset and methods of clustering.

4.1 Dataset

In this thesis we used parts of an open dataset provided by CodeSearchNet [11]. The original dataset is built up of six programming languages containing, most relevant to this thesis code functions and docstring, a short comment on the intended utility of a function, along with

various metadata.

The six programming languages in the dataset are JavaScript, Java, Go, PHP, Python and Ruby.

For the purposes of our thesis we restricted ourselves to only base our experiments on the programming language Python and hence the rest of the dataset was omitted in this thesis.

Number of functions	503502
Number of repositories	12482

Table 4.1: CodeSearchNet Python dataset in numbers.

4.1.1 Search Dataset Summary

The CodeSearchNet dataset for Python consists of repositories publicly available on GitHub, that are not forks, are used by at least one other repository and is licensed to allow redistribution. The following is the methodology that the authors of CodeSearchNet used to create this smaller Python dataset [11]. The functions and respective docstrings were extracted using TreeSitter. Functions without docstrings were removed, functions where the docstring was shorter than three tokens were removed, functions shorter than three lines were removed, functions with "test" were removed, standard extension methods and constructors were removed and finally duplicates were removed. The resulting dataset consisted of 503502 functions, belonging to 12482 repositories.

4.1.2 Selected Parts of Dataset

From the dataset we used the data fields "repo", "url" and "code". The "code" field held the source code block of the function, the "repo" field held the name of the repository and the "url" field held a GitHub url to point to the specific rows where the function source code was found in the repository of a specific version. Notice that since the "url" field held the exact information to find the rows for the function in a specific file it also included the entire file path to the function, from the top-level directory of the repository. We used the "code" to generate our function embeddings and "repo" to split the functions into their respective repositories.

4.2 Repository Search

In order to build a recommendation system or search engine, we need a way to distinguish two repositories apart and then rank them according to the input query which means we also have to be able to make comparisons between natural language and collections of code blocks which is our representation of a software repository.

Central to this thesis is the idea to build upon embeddings of both the natural language query and embeddings of the code blocks. With an embedding representation of both code functions and natural language queries we have a common vector space in which we can compare the two based on their semantic intent.

The next step is to group all of the function embeddings that belong to a certain repository and aggregate them either by conventional methods or by using clustering. The end result of such an aggregation is what we call repository embeddings. If multiple clusters emerge from a repository we imagine that they have gathered together, from single function embeddings, some semantic intent that is important for representing the end use of a repository.

Finally these repository embeddings can be compared to an embedding of a natural language query, and ranked by their distance to it, resulting in a ranked list of recommended repositories to the user of our search engine.

Each of these steps can be done beforehand, making searching efficient as only the embedding of the query has to be determined while searching. Starting from the dataset containing the code functions in plain text, we embedded each code function to an embedding and added it as a new column in the dataset. From here, we aggregated information using methods of clustering and added them as new columns to the dataset, each aggregation method was assigned its own column so that we could differentiate the results between aggregation methods.

4.2.1 Repository Embeddings

To embed the code block for each entry in the dataset we used an open source encoder-decoder model, UniXCoder [9]. The model *API*, application programming interface, has a particularly well suited mode of operation that allows us to map a block of text, in our case the code function, to an embedding, a numerical vector of size 768. The model is also able to utilize the docstring to improve the embeddings of functions, however since we wanted to find out how well this approach would work on source code in general, even where documentation is missing, we elected to not use this feature.

Once using the system the user would enter a query for a functionality in a repository, in natural language. This natural language query, in plain text, would also be embedded using the same *API*.

4.2.2 Determining aggregations

The two algorithms that we used for clustering embeddings in a repository, k-means clustering and Ward clustering, required a target number of clusters to be set. However, since different repositories contained many different numbers of function embeddings and more importantly contained different numbers of separable “functionalities” the best number of clusters could not be known beforehand.

To tackle this problem, the approach that we used was to specify a numeric range of possible different “functionalities” in a package and then cluster the package multiple times. We then estimated the best clustering for each package by the Silhouette Score of each clustering. For the clustering evaluation we used the silhouette score functionality from *sklearn* metrics package [3].

For aggregation methods that did not utilize clustering, namely folder aggregation and repository mean aggregation this method was redundant, as the number of clusters or aggregations was determined without specifying a certain number of clusters beforehand.

4.2.3 Mean aggregation method

Our first approach to a repository embedding was what we called the mean embedding. The mean embedding of a package was calculated as the element wise vector mean of all the embeddings contained in the repository.

Our intuition behind this approach was that if a package provided some "functionality" that abstract notion of a "functionality" would necessarily be built up from the functions in the package and by taking the element wise vector mean of all embeddings we would take all of the functionality provided by the package into account when expressed as a single embedding for the package.

4.2.4 k-means aggregation method

In this approach we used the k-means algorithm for clustering the embeddings contained in a single repository. We used the k-means algorithm provided by the *sklearn* clustering package [4].

The k-means clustering algorithm requires that you specify the number of clusters that you want to divide the dataset into. This meant that we had to estimate a range that the set of embeddings in the package could be clustered into. This range, denoted r, determined from the number of functions, denoted n, in the package was calculated from

$$r = [2, \max(\min(int(\sqrt{n}), 15), 2)]$$

As mentioned before, we had to cluster k-means multiple times to find the optimal number of cluster as determined by the Silhouette Score. This range was just to limit the number of clusterings we did for a single repository to at maximum 15 and at least 2.

In the end we combined the embeddings in each cluster by taking the element-wise vector mean of all embeddings belonging to the cluster. The cluster embeddings were then appended to the dataset as a new data field so that each code-docstring pair in the dataset now also held the information of which cluster it belonged to.

4.2.5 Ward aggregation method

We used the Ward clustering algorithm from *sklearn* [5], [23]. The parameter n_clusters was set in a similar way to that for k-means, with the range

$$r = [2, \max(\min(int(\sqrt{n}), 20), 2)]$$

Afterwards, the embeddings assigned to a specific cluster were aggregated into a single embedding by the mean approach, taking the element-wise vector mean of all embeddings in the cluster and added to the dataset in the same way as in the k-means aggregation method.

4.2.6 Repository File Structure Aggregation

For this aggregation method we changed our approach, instead of finding clusters in repositories by purely numerical methods, we tried to capture if developers of open source were better than numerical methods to group together functionality in the repository.

In this aggregation method we classified the functionality in a repository by the lowest level directory that a function belonged to. It is very common for developers to divide the repository into directories containing functionality commonly used together.

After grouping together function embeddings that were found to belong to the same directory, the mean embedding was calculated among the embedded functions and appended to the dataset for the entries in the given cluster.

Here we decided to go for the lowest level directory name in classifying as it seemed to give a similar number of clusters as the other aggregation methods.

4.3 User Study

In this section we will describe how we conducted the survey, who participated in the study, how data was collected and how we improved the process over iterations. The user study was done during the second half of May 2022.

4.3.1 User Study Setup

For the user study we built an interactive script that let users enter a natural language query to find matching software repositories.

The subjects were first tasked with searching for some functionality that they sought to use and the script would return the highest ranked matches in our dataset of Python repositories using one of our models. The subjects' second task was to annotate the recommendations they received by the script by their relevance to the query. These annotations were explained by the authors to be one of the number 0, 1, 2 or 3. The number corresponded to a description of relevance presented in Table 4.2.

These descriptions were explained by the author before the survey and were available in the survey which can be seen in Appendix A. The collected relevance annotations were what we later used to calculate NDCG for our different models.

4.3.2 Participants

The participants of the user study were people who we contacted directly, either through our own network at LTH or through Debricked. They were asked to participate in evaluating a search engine for open source software repositories in Python. The participants were either professional software developers or students of either Computer Science or Engineering Physics with a focus on software development, machine learning or statistics.

All participants were familiar with Python and had used open source Python software and libraries, but it was not the main programming language used by every participant. Before the survey they knew that the underlying model for recommendations was using functionality search and that this was for our thesis on the subject. They did not know what model they were evaluating and they were not told in what way the annotations would be used to evaluate the model.

Because the data was anonymous we do not have numbers on how many participants there were in total or which queries were annotated by which participant, but from checking logs of who were contacted and notes from the time there were between 15 and 20 participants.

Ranking	Description
0	Totally irrelevant; I would never want to see this for the given query.
1	Weak match; Not exactly what I am looking for but there are some useful elements/pointers to things that I would use and can form the basis for a new query or exploration towards solving my query.
2	Strong match; This does more or less what I was looking for. I could use it as the functionality exists in the package, but the package as a whole does not seem like a perfect fit or the functionality is not generically useful but bound to another specific use case.
3	Exact match; This seems exactly what I was looking for. I would use this package to solve my query (perhaps just exploring a few other options before committing).

Table 4.2: Description of the different relevance scores for recommendations, slightly modified from the relevance score descriptions used in CodeSearchNet [11].

The annotations were not equally split between participants, some only annotated one while others annotated multiple queries.

4.3.3 Survey

The study was first created in a "jupyter notebook", an interactive Python program, where the objective of the survey and all of the instructions was provided to the subject of the survey.

Before the survey was given to the participant we did an initial setup to select the model. The participant was then presented with the survey showing an input box where she could input a natural language query describing the functionality that they looked for in an open source repository. Images of the survey are available in Appendix A

Our system would upon pressing the "search" button query the selected model and return the ten best matching open source repositories for the subject to annotate. Beside the recommended repositories the three best matching functions were also presented to help the participant begin exploring the relevance of the repository.

The script running in the background of the notebook content presented to the subject would then automatically store all of the matched repositories together with the score that the subject provided for each result for later model evaluation. In addition to the relevance scores we presented earlier the relevance score could take a fifth value; -1, meaning the repository was not annotated for the query. This meant that we could later analyze how many surveys were partially annotated and to what degree. The information was saved as a separate csv file for each survey session. In total we collected 64 surveys including the pilot

study. However some were only partially filled in and our annotated dataset only amounted to 49 unique queries, where queries from the pilot study were only included if they had been annotated again after the pilot study.

Later in the study surveys were conducted using a simple website instead, to enable easier sharing of the survey remotely and with subjects outside of the company. The basic functionality and information presented was identical for this version of the survey, and this version can also be found in Appendix A.

Over time we changed the aggregation method for the repository embeddings used in the survey so that we later could compare the aggregation methods against each other. In the beginning the aggregation method was more or less random but towards the end we started to more actively select aggregation methods so that we would get a more balanced distribution of conducted surveys among the different aggregation methods, i.e. we wanted about an equal amount of annotations for each model.

4.3.4 Iterations

The survey went through two iterations, not counting the change in interface from "jupyter notebook" to HTML website.

In the first iteration we returned three recommendations to each subject, as we first thought that overloading the subject with results would take too much time and make for fewer total surveys produced.

However, after a few sessions using this survey we realized that since NDCG requires multiple annotated recommendations for each query we should probably have more annotations per query to increase the probability of annotated repositories ending up as top recommendations for all models. With this realization we changed the number of recommendations from 3 to 10. The first iteration can be seen as a pilot study we used to evaluate the survey setup and most queries annotated by participants during this pilot study was attempted again in the real study. All participants of the pilot study also participated in the second iteration of the survey, but not all participants of the second iteration also participated in the first.

The first, pilot, iteration resulted in 60 annotations while the second resulted in 440 annotations for a total of 500 annotated repositories. This said, there was a lot of overlap between the queries and repositories of the iterations so in effect only the 440 annotations from the second iteration of the study were used.

4.4 Evaluation

To determine the best approach of producing functionality embeddings, we used two methods of evaluation, an internal statistical evaluation and a user study.

4.4.1 Internal Statistical Evaluation

Our first evaluation of the models was a statistical evaluation. For each software repository we built datasets of repository embeddings and cluster embeddings and evaluated the results using the metric Silhouette Score. To compute this metric we again used the *sklearn* metrics package, which we also used when creating the clusters [3]. Silhouette score for each

model was needed to be able to analyze the connection between the separation of clusters and relevance metrics to answer **RQ4**.

4.4.2 User Study Evaluation

Finally, after gathering the user study data to an annotated dataset, we evaluated our different models' ability to recommend relevant repositories using the dataset.

We calculated NDCG for each of our models by querying each model with each of the annotated queries and then computing the NDCG for the recommendations. The mean of the NDCG for the queries was the evaluated model's final NDCG.

Each model was also evaluated on how well it compressed the search space, as we go from a number of functions to a handful of repository embedding we effectively lower the number of comparisons that is needed for each search in the system. In theory this will lead to faster search times for a given query.

Chapter 5
Results

In this chapter we will present the results of the work done in this thesis.

In the first section we present the results of the user study that was conducted for this thesis. Conducting a user study was a way for us to produce meaningful measures of how useful such a system could be to real end users. Here we present raw statistics of the resulting dataset that was built up from the annotated results of our participants.

Following our user study results we analyze the dataset containing our repository embeddings. We provide the reader with detailed information about how silhouette scores varied between clustering methods as well as how different methods of clustering lead to differences in numbers of clusters over the dataset. This section gives us particularly useful information in our aims of answering **RQ2**.

In the third section we analyze the results of the user study in detail, and use the annotated data to evaluate and compare our methods. We then explore altered results with the ambition of removing bias from our survey results. Of special interest in this section is Table 5.5 in answering **RQ3** as well as Table 5.4 in answering **RQ2**.

We end this chapter with giving an overview where our system failed to produce any relevant information. The examples in this section are to be contrasted with the one given in the first section to show that not all results were satisfactory to our participants, and to be used as a basis for how our system can be improved, more on which will follow in the next chapter.

5.1 Survey

In this section we will go over the results from the user study. We present the dataset that was created from the annotated recommendations from our participants' usage of the search engine as well as give the reader some examples of particularly well performing queries.

5.1.1 Survey Data

The collected survey data is a number of csv files containing queries, repositories, the position our chosen model gave the repository and finally the score given by the human subject to that repository for the search query. In the survey data the "OPTICS" model is mentioned for collecting annotations, first in Table 5.1. OPTICS is a clustering model that we tried during the user study to collect annotations, with the goal of also comparing it to the other models. We realized however that we might not have used it correctly and as such did not want to include it in the comparisons, however as this should have no effect in using the annotations from sessions with this aggregation model we decided to still keep them in the annotated evaluation datasets.

Model	Annotated Repositories	Annotated Queries
Annotated queries using Ward	132	23
Annotated queries using k-means	120	12
Annotated queries using Folder	98	13
Annotated queries using OPTICS	150	15

Table 5.1: Distribution of annotated queries and repositories for the different aggregation models.

A discrepancy can be seen in Table 5.1 where Ward has many more annotated queries despite not having many more repositories. This is because a few of the earlier survey sessions ended without all repositories being annotated.

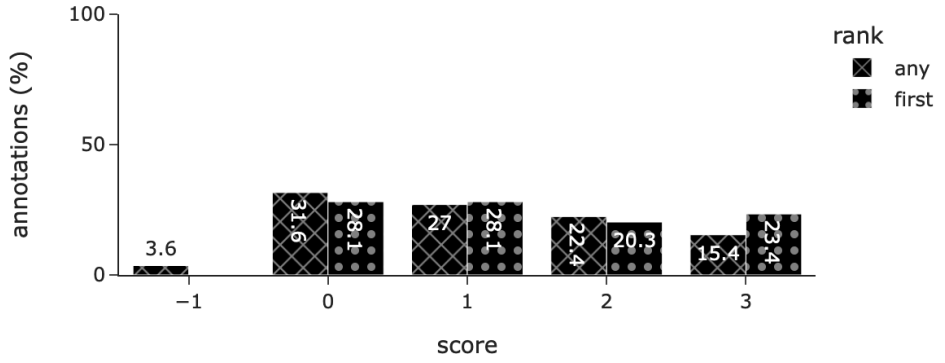

Figure 5.1: Distribution of relevance scores in general and for first rank for the clean dataset.

In Figure 5.1 an overview of the annotated data can be seen. An interesting note is that there are fewer annotations with higher relevance scores, as opposed to the annotated dataset in CodeSearchNet which was more balanced.

Finally, because we found some issues with the annotated dataset we created a subset which we call the "clean" dataset where we removed these issues. The way we decided which queries to keep was through the following criteria; there should be at least 10 annotated repositories for the query and the query should not be a duplicate. The first criteria was needed because of queries that were only annotated in the pilot study where 3 repositories were recommended and where a survey was given up halfway through. The second criteria

was needed because it happened a few times that participants saved a completed a survey but realized they did a mistake, and then re-annotated the same query once more. We call the complete annotated dataset "raw", because we did not edit anything.

Model	Annotated Repositories
Annotated queries using Ward	88
Annotated queries using k-means	90
Annotated queries using Folder	73
Annotated queries using OPTICS	110

Table 5.2: Distribution of annotated repositories for the different aggregation models in the clean dataset.

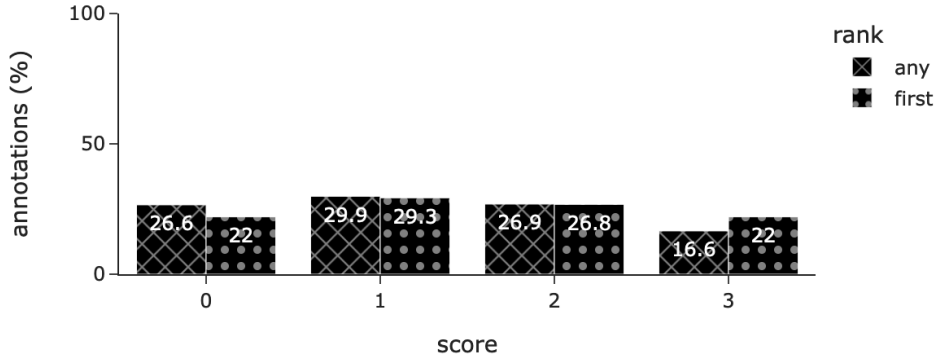

Figure 5.2: Distribution of relevance scores in general and for first rank for the clean dataset.

A summary of the distribution of relevance scores in the clean dataset is presented in Figure 5.2, and the distribution of annotated repositories by aggregation models is presented in Table 5.2. Notable in both these tables is that the distribution is quite similar, but less skewed towards zeroes. The clean dataset is still comparably balanced between the different aggregation models, so it should not benefit any specific model.

5.2 Clustering Metrics Overview

To answer **RQ4** of whether separated clusters correlated with relevant recommendations we needed to calculate how separated our clusters were. For all clustering algorithms the goal was to maximize Silhouette Score, while if possible also maximizing number of clusters, i.e. if k-means creates clusters with two different values for n but they both achieved the same Silhouette Score, the clustering with the larger n value would be preferred. The only exception was "folder" where the folder structure of the repository was used instead of unsupervised clustering. The mean number of clusters and mean number of clusters of repositories with more than two functions for each clustering algorithm is presented in Table 5.3. For a better understanding of the Silhouette Score we present the distribution of Silhouette Score between clusters in Figures 5.3, 5.4 & 5.5, this shows that most clusters have a Silhouette Score close to the mean and that there are very few extreme outliers.

Model \ Metric	Mean Clusters (Total)	Mean Clusters (> 2 functions)
k-means	3.05	8.75
Ward	2.34	13.3
folder	2.34	14.1

Table 5.3: Mean number of clusters for the different clustering methods tested in the study.

Interesting to note is that there is no model that produces more than 2 clusters for a majority of repositories, which is seen in the column named "% > 2 clusters" in Table 5.3. Also bear in mind that cluster number for k-means and Ward was artificially capped due to computation constraints, so the numbers in Table 5.3 should not be read as general for the clustering algorithms on this dataset but rather as data on the clusters we worked with.

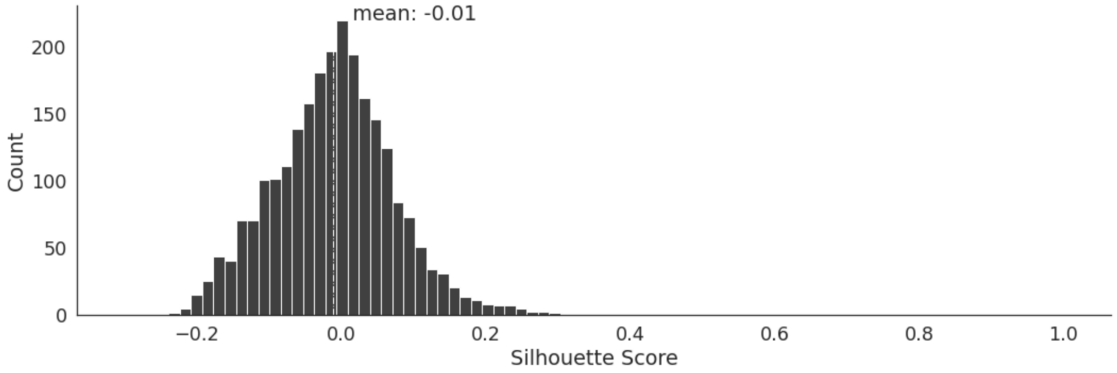

Figure 5.3: Folder Silhouette Score distribution.

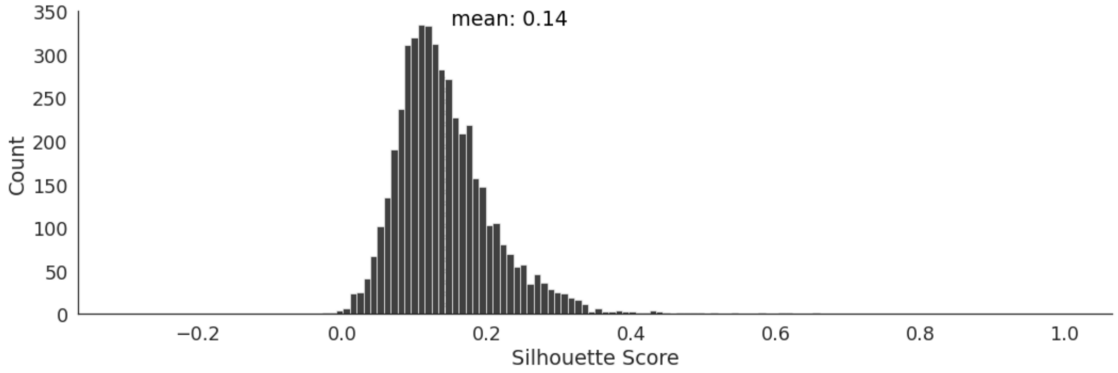

Figure 5.4: k-means Silhouette Score distribution.

The distributions in Figure 5.3, Figure Figure 5.5 and Figure Figure 5.4 shows that the Silhouette Scores for neither of the clusterings have many outliers. The Silhouette Scores indicate that the clusterings for "Folder" are not very separate, since the score is very close to 0. The scores for k-means and Ward are slightly higher, but still not close to 1. This indicates that the clusters are not very well separated, but that the separation is better for Ward and best for k-means.

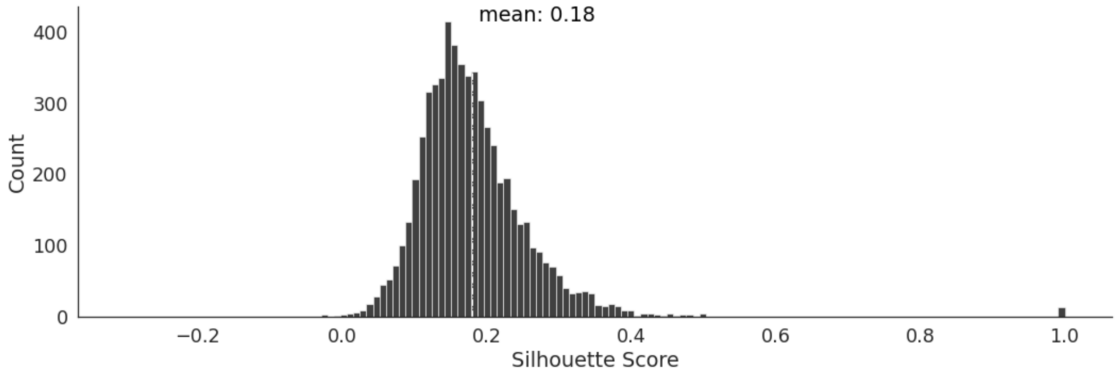

Figure 5.5: Ward Silhouette Score distribution.

5.3 Recommendation Metric Comparison

To answer **RQ1** and find out if clustering of function embeddings could be used for functionality search of software repositories, we measured the NDCG of the different models to compare them to a baseline of just searching using the function embeddings. In Table 5.4 the models respective NDCG score, the relevance of recommendations, are compared. We chose to set the k value, number of recommendations returned for a query, to 10 because this is a common number of recommendations and specifically what was used in CodeSearchNet and our survey. Since we also wanted to look at scalability of the models to answer **RQ3** the same values are presented in Table 5.5, but where NDCG scores are divided by space utilization to also factor this in to the comparison.

After compiling all results from running the NDCG evaluation script with each model on the collected survey data, we created tables for comparison between the different approaches. The compiled data is separated into three groups. In the first one we present the results on information retrieval and search space utilization for aggregation methods alone. In the second one we present the results of information retrieval when pairing our aggregation methods with function embeddings, contextualizing the function embeddings. In the third one we present the results of information retrieval when contextualizing the function embeddings with multiple aggregation methods.

5.3.1 Aggregation

In Table 5.4 the NDCG results of all different aggregation models are presented.

In Table 5.4 the highest NDCG scores, 0.63 and 0.57, is for just using function embedding. The score indicates that the model often recommended repositories with high relevance from the annotated ones for a given query and that it ordered them quite close to their order of relevance, although not perfectly. The results also indicate that k-means does give recommendations almost as relevant, that Ward is just slightly worse and that Folder is recommending less relevant repositories than either of the clustering algorithms. Overall these NDCG scores indicate that either aggregation model is able to recommend mostly relevant repositories given that the annotated dataset is accurate.

In Figure 5.6 the different size of the search space, or number of vectors to search through for a query, is shown as a percentage of the full dataset which included over 500000 functions.

Aggregation \ $NDCG_{dataset}$	Raw	Clean
Embedding	0.63	0.57
Repository Mean	0.46	0.37
Folder	0.48	0.41
k-means	**0.55**	**0.52**
Ward	**0.55**	0.49

Table 5.4: NDCG scores for different aggregation methods. Baseline scores are above dotted line and best non-baseline scores are in bold font.

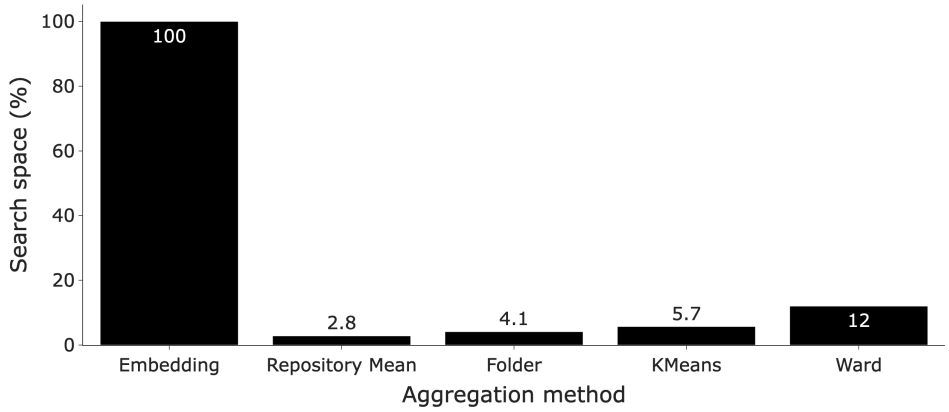

Figure 5.6: Search space of the different aggregation methods compared as a percent of the full embedded dataset.

Interesting points in this comparison is that Folder and k-means have quite similar search space utilization and that both use less than half of the search space compared to Ward. This showcases that the difference is substantial and could potentially affect both speed of search and memory consumption for an embedding search model. For Table 5.5 the NDCG scores from Table 5.4 are divided by the percentages in Figure 5.6 to make a comparison which takes both recommendation relevance and scalability into account.

When scalability or dataset size is taken into account, as is shown in Table 5.5, using the repository mean as an aggregation method outperforms all other methods, but both Folder and k-means are also an order of magnitude better than using functions embeddings as search space.

5.3.2 Weighing Embeddings & Aggregation

The scores received when using a mix of cluster embedding and function embedding are seen in Table 5.6. We tested a 50/50 mix between function and cluster embedding because to our knowledge there is no specific reason to try another division first. That said, it could be that something like 10/90 or 70/30 would be better, but that is left for future study since we are just interested in finding out whether the cluster embedding could improve upon only using function embeddings.

Aggregation \ $\frac{NDCG}{SearchSpace}_{dataset}$	Raw	Clean
Embedding	0.63	0.57
Repository Mean	16.4	13.2
Folder	**11.7**	**10.0**
k-means	9.6	9.1
Ward	4.6	4.1

Table 5.5: NDCG scores divided by Search Space Utilization as a combined score. Baseline scores are above the dotted line and best non-baseline scores are in bold font.

Aggregation \ $NDCG_{dataset}$	Raw	Clean
Embedding	0.63	0.57
Folder	0.7	0.65
k-means	**0.73**	**0.66**
Ward	0.67	0.63

Table 5.6: NDCG scores for the models, where models are 50/50 between model and function embedding. Baseline scores are above dotted line and best non-baseline scores are in bold font.

As is seen in Table 5.6 combining cluster and function embedding does improve upon using only function embeddings regardless of clustering used.

5.3.3 Reducing Bias

Due to the possibility of annotations collected with a specific model biasing the result of calculating NDCG for that same model we elected to also run the NDCG evaluation for the models while filtering out the annotations provided when using the model for surveys.

In Appendix B we present some data on our attempt to reduce bias. The results indicate that our methods of reducing bias does lower the overall NDCG of the evaluated models, meaning that recommendation relevance is probably lower than Table 5.4 indicates, but that the comparative scores are to a large degree not affected, e.g. k-means is still the best performing aggregation model for recommendation relevance and still compares well to using function embeddings directly. For a comparative view of how the methods of reducing bias affects the results see Table B.7 in Appendix B.2.

5.4 Correlation Between Clustering and Recommendations

To answer **RQ4** and determine whether using better clustering, measured by Silhouette Score, would lead to better search results, measured by NDCG, the correlation between these metrics for our study is presented in this section.

The measurements of correlation between Silhouette Score and NDCG are based on the same values presented in Table 5.4 as well as Figure 5.3, Figure 5.5 and Figure 5.4. The results of calculating the correlation between Silhouette Score and NDCG are presented in Table 5.7. These values were calculated using the function "corrcoeff" in the "pandas" Python framework which calculates the Pearson correlation coefficient.

Dataset	Correlation
Clean	0.94
Raw	0.98

Table 5.7: Correlation coefficients for the correlation between Silhouette Score and NDCG.

The correlation coefficients in Table 5.7 seem to indicate a correlation between Silhouette Score and NDCG. Note that the coefficients in Table 5.7 are based on a small number of data points.

5.5 Where the method fails

Going through the results we were able to group together similar cases where our method failed to deliver desirable results. The most common of these failures by the model were queries which included more than one specific functionality. Table B.3 presents a few examples of this. These failures can be used to get an insight on the limitations of this approach and help answer **RQ2** on the degree of which using aggregation methods for functionality search is viable.

We also found that the model particularly struggled with uncommon words. A notable query which included what the authors deemed an uncommon keyword and which resulted in poor quality recommendations is "elo rating python", which achieved the low score of **0.5** out of a possible **31.8**. We believe this may be due to the language model not understanding the word "elo" and that it highlights a shortcoming of the approach, namely that there is a possibility that the embedding model has not seen specific words often enough to properly place text including them correctly in its vector space.

Chapter 6
Conclusion

In the first part of the conclusion we aim to answer our research questions from our results as best as we can. We argue that embeddings is a viable way of finding relevant software repositories from a natural language query and that aggregating data into repository embeddings can in fact lead to better performance than the naive approach of using only embeddings, and that aggregating has the additional benefit of scaling better. We also find that separation between clusters and relevant recommendations seem correlated.

Following our research question we continue the discussion around several key topics related to this thesis, such as limitations, ethics and how our results compare to CodeSearchNet.

In the third section we present some potential threats to validity to this thesis, especially with regards to our mistakes in human evaluation. With all user studies there is always a risk of biasing the participants, and in hindsight many things could have been done to minimize such bias. We also touch on our selection of clustering algorithms and end this section with a discussion on how the scope of our thesis limits the potential generalized benefit of the work.

We end this chapter by presenting ideas of future work related to this thesis and some final words.

6.1 Answers to Research Questions

6.1.1 RQ1

Just as earlier research suggested, we found that utilizing source code embeddings in the domain of semantic search does work, and that it can be utilized with promising results even on the abstraction level of repositories.

The exact degree is harder to tell, but it does seem like using function embeddings does provide mostly relevant searches while there are still many highly ranked recommendations that are also irrelevant to the query. A good indication of this is Table B.1 where we can

see that multiple queries got very relevant recommendations from the tested models. It is harder to use NDCG to evaluate absolute performance rather than comparative, but looking at Table 5.4 and Table 5.6 as well as the bias adjusted Table B.6 we can see that for using only embedding all NDCG values are at least 0.40, which does indicate high relevance.

We did observe various problems with the approach, but these issues could potentially be solved by using functionality search on top of more common recommendation systems, such as keyword search combined with different filtering constraints like popularity and usage statistics. The biggest problem is visible in Table B.3 where we can see poor relevance results found during our human evaluation. What these seem to indicate is that this model cannot handle multiple separable functionalities, e.g. a query describing both read and write functionality will not receive very relevant results. This will be further discussed in section 6.2.

6.1.2 RQ2

The answer to **RQ2** on to what extent aggregations of function embeddings can be used to find relevant repositories we look at how the aggregation methods compared to directly using function embeddings and the achieved NDCG when combining the methods.

Using only aggregation methods on their own does not really compare favourably to using function embeddings directly. But that said, both of the tested clustering algorithms tested as aggregation models achieved mostly scores similar to function embeddings, differing at most 0.14 which is seen in Table B.5 on the raw dataset between function embedding and the Ward clustering algorithm.

Where the aggregation methods compare well to just using function embeddings was when used together with the specific function embedding. Using aggregation in combination with function embedding achieved the highest NDCG scores we found, 0.73 for a 50/50 combination with k-means. This indicates that aggregation can improve relevance in search recommendations when compared to just using function embeddings directly.

Overall we conclude that aggregation methods can be helpful for the task of finding relevant repositories and to a greater extent can be used to improve upon using only function embeddings.

6.1.3 RQ3

While utilizing function embeddings directly does provide more accurate results, the most accurate being a combination of cluster embedding and function embedding in our research, it does come with the cost of a less scalable model. As seen in Figure 5.6 the space requirements are 6 to almost 25 times as high for storing all function embeddings compared to storing only the aggregated embedding. Another scalability benefit of using the clustering model is that we can decide on a maximum number of clusters to divide a repository into.

We do believe that scalability is an important factor for this problem, as the number of open source software repositories is great and since this number is ever growing, as of writing these words GitHub is hosting over 330 million repositories and if the same ratio between repositories and and functions as seen in Table 4.1 holds true that would translate to over 12 billion functions. Therefore, a model which is able to scale is probably preferable in this domain. Finding a clustering method that provides accurate categorization which helps in the

domain of functionality search is therefore still of importance, even though purely searching for functions might be more accurate.

The most scalable model is as discussed before to simply take the function embedding mean of the software repository, but it is also worth to note that both k-means and Ward are space efficient and get much higher NDCG scores.

6.1.4 RQ4

From our very limited set of aggregation models and data points the general trend seems to be that Silhouette Score and NDCG are correlated. In Table 5.7 the computed correlation coefficients are 0.94 and 0.98. As a first trial this does seem to indicate that there is a correlation, but due to only trying 3 different aggregation models with different Silhouette Scores this is far from conclusive. Further work is needed to decide if this is really the case. If we assume that the result is reliable then the answer to **RQ4** would be that there is a strong relationship between aggregation models achieving clear separation and being able to recommend relevant repositories.

6.2 Discussion

We have researched whether unsupervised clustering and other aggregations methods can be used to improve information retrieval and scalability for functionality search of software component search.

Introducing a combination of aggregated embeddings and the functionality embedding or using the aggregated embeddings alone, can both increase the accuracy of search results and improve scalability of a recommendation system respectively.

6.2.1 Limitations of the clustering approach

There are of course many limitations of the clustering approach for functionality search, one large one being the dependence on good function embeddings. The performance of this model cannot be good if the function embeddings do not accurately match their function's semantics.

Another limitation that we found while analyzing the data was that the model seems to struggle when multiple functionalities are mentioned in the same query, such as the example of "read and write metadata from image files" which can be seen in Table B.3. Here we believe the model recommends irrelevant repositories because there are individual embeddings for functions that either read or write image metadata, but not individual functions that do both. This then means that we might recommend repositories which read image metadata or that creates images and thus writes image metadata, but are unlikely to recommend a repository that edits image metadata.

6.2.2 Ethics

In this study we have not been particularly concerned with ethics, but potential concerns are mainly with bias and fairness.

If a software component search engine using clustering of function embeddings would be popularized it could be biased towards certain programming languages and programming styles. Since we want function embeddings that are not biased towards language or style, but rather embeddings that only represent the semantic intent of the functions, this should not be a problem. It is still worth consideration since the underlying language model used for embeddings could be biased in such a way.

Another concern is that, if popular enough, it could potentially impact how developers write software. This would be similar to how websites operate today, where the content needs to be written in such a way as to be given a high recommendation in Google's search algorithm, so called "search engine optimization". An example would be if components started to include functionality for a specific popular task simply to end up recommended for more queries.

6.2.3 Comparison to CodeSearchNet

While CodeSearchNet and our works differ in what is being recommended it is still interesting to see what other NDCG scores are considered valuable in the field of functionality search. The highest score achieved on the CodeSearchNet leaderboard was 0.47 for Python. While it might seem that our model generally performed better than this, there are a few things to keep in mind. CodeSearchNet had a much larger annotated dataset, 2089 annotations for Python, and that dataset differed from ours in one key aspect; the annotations were not collected by participants who tried a search and annotated probable relevant candidates but instead annotated functions from a small dataset that had no reason to be either relevant or irrelevant, and the participants neither expected relevant or irrelevant results. This probably reduced their bias towards too high NDCG scores and is something we discuss in threats to the validity of our work.

6.3 Threats to Validity

There are multiple threats to the validity of the results and conclusions in this thesis, mainly due to the nature of surveys and human evaluation being a difficult way to acquire unbiased data.

6.3.1 Search Dataset

For the search dataset we utilized the same one used by CodeSearchNet functionality search for the Python programming language. This was the dataset where tests, short functions and functions without docstrings were filtered out. However, CodeSearchNet also provided another dataset with all functions which was about four times larger and could have been combined with the function-docstring dataset for even more functions. Since we did not use the docstrings anyway we could have utilized this larger dataset which may have provided better data for clustering due to having more functions per repository on average. It could also be that utilizing this larger dataset would have resulted in worse performance due to having more functions with less information and more noise in general. In either way more

data would have enabled more accurate results, and in hindsight we should have used the larger dataset.

6.3.2 Mistakes in human evaluation

One mistake during human evaluation or surveying was some ambiguity in how subjects were to evaluate the results. The results were explained to be an as good match for the query as possible, e.g. the repository should be related to the functionality described in the query and solve the problem facing the subject. Some subjects believed that a repository simply having a function matching the intent of the query was enough to warrant a high score in evaluation - which is a failure of the authors for not providing less ambiguous instructions. The result of this mistake might be a bias towards the models using function embeddings which should be kept in mind when looking at Table 5.6. But this can also have introduced a positive bias overall, meaning we might be seeing higher NDCG scores than deserved for all models.

Another mistake made during the user study was that results were presented in the order they were recommended by our model. This introduced a bias in our subject that the earlier a repository appeared in the list of results the more relevant a recommendation it "should" be. This was not mitigated in any other way than that we told the subjects not to care about the order. An approach that reduces this bias would have been to always randomize the order, but that is unfortunately not something that was done for the survey data collected in this study. This is probably introducing a positive bias for all models resulting in higher than deserved NDCG scores.

Another problem in the evaluation was the number of results. Although not confirmed it could be the case that less time was devoted to evaluating recommendations appearing later in the list of results, simply due to the fact that subjects got tired of evaluating after analyzing a couple of repositories. This bias towards time spent on higher ranked recommendations could also have been mitigated by randomizing the order of recommendations presented to subjects or by having fewer recommendations to rank per query/survey session.

Finally we found that collecting our annotated dataset in surveys using our different models was probably the wrong approach and that either just using function embeddings during the surveys to not bias between the aggregation models. Even better would be doing as CodeSearchNet did for the dataset and giving the human evaluators the task of annotating a number of items for a query without having the subject come up with the query and selecting the items for annotation without the models we later want to evaluate, e.g. selecting a random set from the search data set. Since the participants would then not be expecting especially relevant results this would probably affect bias towards higher relevance scores. Since our participants knew us, wanted us to succeed and knew they tested our models, they might have scored the relevance of the recommendations higher than deserved and thus our NDCG scores are probably higher than they should have been. Using this approach for annotation collection would have the added benefit of enabling us more control in balancing the data set. There would not be specific queries with bias towards a specific model. This is something we tried to adjust for in Table B.5, but of course this also reduced the size of the already small data set which makes the results less reliable and is thus not an ideal solution.

While the scores our models achieved in this study are probably higher than deserved due to our biases we still believe the comparison between models, especially when looking at Table B.7 where we present different approaches for reducing bias, should be somewhat reli-

able. This means that although we might not be able to claim in absolute terms to what extent the models produce relevant recommendations, we can say that using clustering algorithms compares well to just using function embeddings directly.

6.3.3 Clustering Algorithms

We evaluated just two clustering algorithms as aggregations in this thesis. The number of data points is thus very low. For the answering **RQ2** this is probably not a big problem as we don't need to evaluate a large number of clustering algorithms to test if the idea of using them for functionality search is plausible. It is however a problem when trying to see if Silhouette Score can be used to select clusterings for functionality search by measuring its correlation with NDCG. In hindsight we should have saved the discarded clusterings with lower Silhouette Scores for the chosen algorithms as well to have more data points when calculating the correlation. Another problem with the clustering algorithms is that they achieved quite low Silhouette Scores in general, meaning that they were not able to find clear clusters or classes of function embeddings and that there is some uncertainty on whether the embeddings were placed in the "correct" cluster. This could indicate that the underlying embeddings were not well-suited for unsupervised clustering. However, since the search relevance results generally seem positive this probably does not invalidate the hypothesis that clusterings can be useful for functionality search for repositories.

6.3.4 Scope

There are also some threats to the validity of this work in the form of the scope of the thesis. Due to focusing on just one programming language, Python, we cannot say whether this is applicable to other languages as well, and if not then the research might not be very valuable. The same goes for users. Maybe the participants in our user study differs a lot from the average software developer and scored the recommendations higher or lower than the industry average. This could also be the case for our repositories. Maybe something inherent to our small dataset of repositories led to the results. One example would be if the authors of the repositories were better than average at writing comprehensive code, which might lead to more accurate embeddings and thus higher than expected relevance score for our models.

6.4 Future Work

Our study looked at a few ideas to utilize function embeddings generated from large language models to help in software component recommendations for search queries. There are many more ideas to explore in this area and many ways to extend the methods described in this thesis. Some future work we thought about while writing this is presented below.

6.4.1 Program Language Semantics

Using program language semantics has a lot of potential to improve the accuracy of our model. Mainly because program language semantics could be utilized to find the most important

functionality in a given repository, which could be used to find what different functionalities a repository actually contains. An example would be that most large repositories contain lots of helper functions, tests and utilities which are not really part of the repository's domain. This is something that tools in the program language semantics area can solve and help our repository recommendation.

One example of program language semantics helping filter out functions that are not useful for repository functionality is simply not including tests, which is what was done in the dataset we used from CodeSearchNet. Tests may hold important information about how the functions are used and what they are supposed to do however. Maybe instead of discarding the tests completely they could have been used to find which functions were most important in the repository and more importantly how the functions are used. If this could be used to further enrich function embeddings, for example so that functions with similar tests are closer together and vice versa, it could probably help in the separation of clusters and in other words make it easier to find different functionality inside of the repository. This is probably the most promising thing to test next as it would be quite easy to get the data and connect tests to their tested function, although not trivial, and we believe tests can potentially say a lot about the semantics of a function.

Another example is using the program's call graph to remove helper functions which are never directly executed by a user of the repository and thus does not represent user functionality in a repository. There are two problems with using call graphs for this however. The first problem is that call graphs are rarely exact, as we usually cannot perfectly determine outcomes of a program without executing it. This is not a huge problem as we can still probably filter out a great portion of non-user functionality despite not having exact call graphs. The second problem is easier to solve, but still needs to be mentioned. To create the call graph we require the full program source code and structure, not just functions in a vacuum. Thus a new dataset would be needed or one would have to use public repositories directly. In the same way as tests can help in finding similar functions and separating less similar ones, we believe the call graph for a function can say a lot about the semantics of the function.

One last example is only looking at public functions, as the private functions are probably not intended for an end user of a repository and should therefore not contribute to the functionalities of a repository.

6.4.2 Embeddings

One way to improve the accuracy of our model is to improve the accuracy of the underlying embeddings. Using embeddings that more accurately represents the semantic functionality of the source code and text that describes it, should translate to more relevant recommendations. We could potentially try using a different vendor or project that is the current or future state of the art, instead of Microsoft's UnixCoder, which should improve the relevance of our models as well.

6.4.3 Clustering on the full dataset

A possible improvement could be to create clusters on the entire dataset instead of per repository. That is, using unsupervised clustering to find all functionality categories in all repos-

itories. We could then label each repository by which functionality clusters it has functions in, and use that information to recommend relevant software repositories.

This approach could potentially be more powerful, especially for smaller repositories where we could only find one cluster which is essentially equivalent to the mean embedding of that repository. Instead of recommending a repository like that by its mean, it would instead be recommended based on what functionality cluster it fit inside of.

The problem with this approach is that for multiple smaller repositories which would be categories in the same functionality cluster we would not, by using only this model, be able to differentiate between them. This could of course be solved by utilizing another data point such as the mean embedding of the repository, best function match, metadata or something entirely different.

It would be interesting to study how utilizing this approach would compare to the one we researched in this thesis.

6.4.4 Evaluation

During the course of the user study, and while evaluating our results, we learned a lot from the mistakes we made and came to many conclusions on how we could have improved this process. Some of these ideas will be discussed here.

The first thing we would change for the user study would be to not collect the annotations through the models we wanted to evaluate, but rather emulate the process in CodeSearchNet and randomly show participants a query and a repository and have them evaluate the relevance. This would reduce many of the biases we potentially had in our results.

Another improvement of the study could be to have a smaller test-dataset where we have good meta information, such as features and functionality, which could be used specifically for evaluation. This would reduce the number of unclear repository recommendations and improve reliability of the results.

Finally a large improvement would be the number of participants and total number of annotations. Increasing the number of sessions from 50 to 100 or even a 1000 would increase the reliability of the results a lot and would hopefully increase the difference between the approaches and allow us to be more certain about our conclusions. If we were to conduct the study again we would want to increase the pool of participants and set aside more time for the data collection since surveys are quite time consuming.

6.5 Final Words

Overall, we find the results of using clustering algorithms to separate and search for functionality in open source software repositories promising, but further study is required and many questions remain. If we were to implement this kind of search model today, we would use the k-means clustering algorithm because it performed very well with regards to NDCG, is space efficient and is a clustering algorithm that is generally resource efficient in terms of computation.

There are both many ways to improve upon our study due to shortcomings and mistakes, and ways to take our models further with new approaches. With these improvements we

believe clustering of source code functionality embeddings can potentially be a powerful and scalable way to help find relevant open source software repositories.

References

[1] About github. https://github.com/about. Accessed: 2023-02-14.

[2] Codesearchnet by github. https://wandb.ai/github/CodeSearchNet/benchmark/leaderboard. Accessed: 2023-02-14.

[3] Kmeans documentation. https://scikit-learn.org/stable/modules/generated/sklearn.metrics.silhouette_score.html. accessed: 2022-09-30.

[4] Kmeans documentation. https://scikit-learn.org/stable/modules/generated/sklearn.cluster.KMeans.html. accessed: 2022-09-30.

[5] Kmeans documentation. https://scikit-learn.org/0.15/modules/generated/sklearn.cluster.Ward.html. accessed: 2022-09-30.

[6] *Oxford English Dictionary Online*. Oxford University Press., 2023.

[7] Egor Bogomolov, Yaroslav Golubev, Artyom Lobanov, Vladimir Kovalenko, and Timofey Bryksin. *Sosed: A Tool for Finding Similar Software Projects*, page 1316–1320. Association for Computing Machinery, New York, NY, USA, 2020.

[8] Noam Chomsky. *Syntactic Structures*. Mouton de Gruyter, 1957.

[9] Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. Unixcoder: Unified cross-modal pre-training for code representation. *arXiv preprint arXiv:2203.03850*, 2022.

[10] Julia Hirschberg and Christopher D. Manning. Advances in natural language processing. *Science*, 349(6245):261–266, 2015.

[11] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. Codesearchnet challenge: Evaluating the state of semantic code search. *CoRR*, abs/1909.09436, 2019.

[12] Kalervo Järvelin and Jaana Kekäläinen. Cumulated gain-based evaluation of ir techniques. *ACM Transactions on Information Systems (TOIS)*, 20(4):422–446, 2002.

[13] W.C. Lim. Effects of reuse on quality, productivity, and economics. *IEEE Software*, 11(5):23–30, 1994.

[14] Christopher D Manning. *Introduction to information retrieval*. Syngress Publishing, 2008.

[15] W. McKinney. Data structures for statistical computing in python. 2010.

[16] Sutskever I. Chen K. Corrado G. S. Mikolov, T. and J. Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, (2013).

[17] Pierre Nugues. *An Introduction to Language Processing with Perl and Prolog*. Springer-Verlag Berlin and Heidelberg GmbH Co. K, 2014.

[18] Peter J. Rousseeuw. Silhouettes: A graphical aid to the interpretation and validation of cluster analysis. *Journal of Computational and Applied Mathematics*, 20:53–65, 1987.

[19] G. Salton and M. E. Lesk. The smart automatic document retrieval systems—an illustration. *Commun. ACM*, 8(6):391–398, jun 1965.

[20] Susan Elliott Sim, Medha Umarji, Sukanya Ratanotayanon, and Cristina V. Lopes. How well do search engines support code retrieval on the web? *ACM Trans. Softw. Eng. Methodol.*, 21(1), dec 2011.

[21] Vijayan Sugumaran and Veda C. Storey. A semantic-based approach to component retrieval. *SIGMIS Database*, 34(3):8–24, aug 2003.

[22] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008, 2017.

[23] Joe H Ward Jr. Hierarchical grouping to optimize an objective function. *Journal of the American statistical association*, 58(301):236–244, 1963.

Appendices

Appendix A
Visual of Survey used in study

A.1 Jupyter Notebook

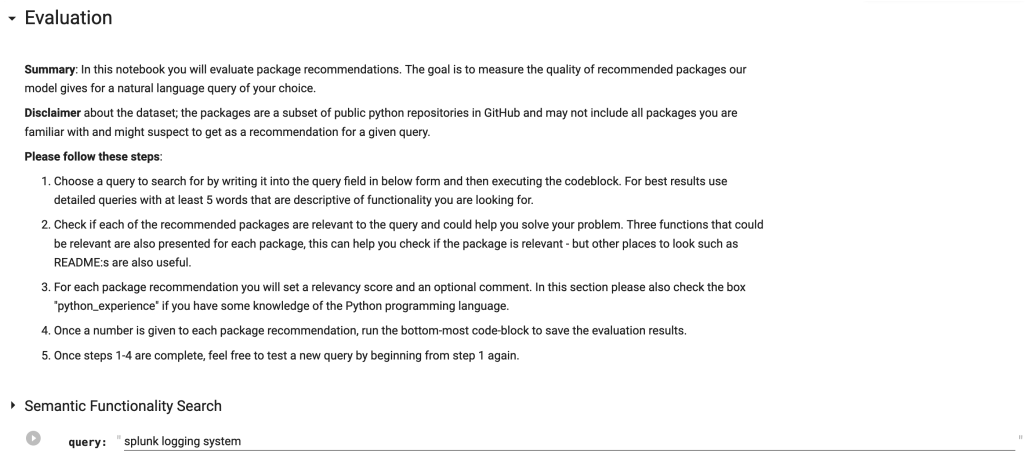

Figure A.1: Beginning of Survey in Jupyter notebook form, where subject is given necessary information to begin the evaluation.

A.2 HTML Website

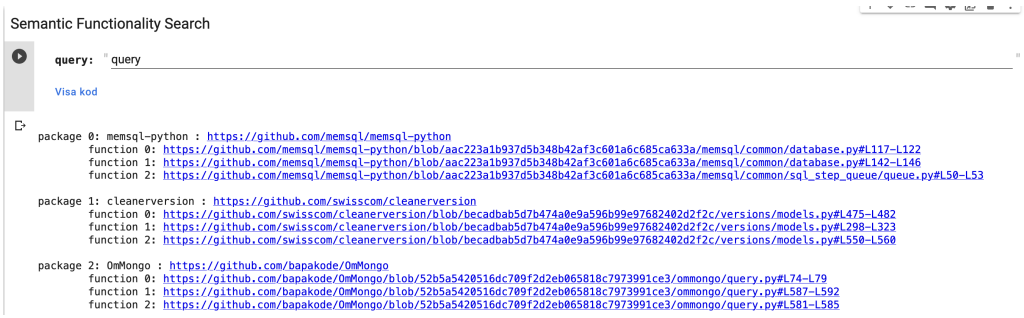

Figure A.2: How the subject where visually presented the results of their search query.

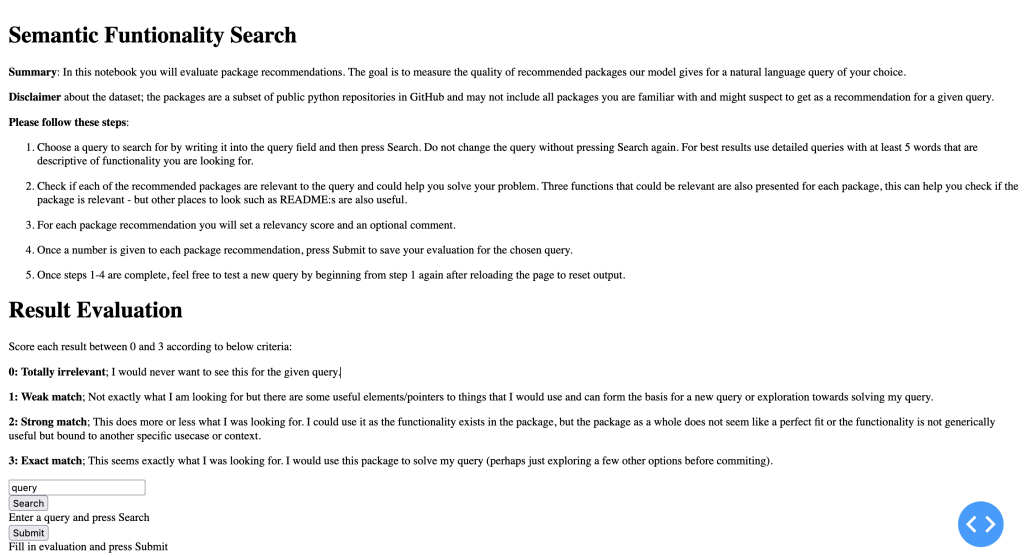

Figure A.3: Beginning of Survey in website form, where subject is given necessary information to begin the evaluation.

Figure A.4: How the subject where visually presented the results of their search query in the HTML Dash website survey.

Appendix B
Additional Results

B.0.1 Queries with best results

The best results using relevant recommendations in the top 3 ranks, i.e recommendations evaluated as a 3 as top results, can be seen in Table B.1.

Query [Model] \ Position	1st	2nd	3rd
“encode string with utf-8” [Ward]	3	3	3
“Create unicode animations in a terminal” [KMeans]	3	3	1
“create simple line plot” [Ward]	3	2	3
“create contextual embeddings from strings” [Ward]	3	2	3
“setup a web server” [folder]	2	3	2
“sent email to users based on their profile” [Ward]	2	3	2
“compress file to common formats” [Ward]	2	2	3
“find dependency relations in pip project” [folder]	2	2	3

Table B.1: User study queries resulting in recommendations receiving high relevance scores in survey, with the scores for the first three positions in recommendation displayed.

Note that the examples in Table B.1 is not an exhaustive list of queries receiving top valuations but rather selected for having a sum of over 6 for relevance score in top three recommendations. The results in Table B.1 use a 50/50 mix between aggregation and function embedding.

In Table B.2 notice that we chose to use DCG for the comparison of these results instead of NDCG, and that a perfect score for DCG would be that 10 relevant repositories annotated as 3s were recommended resulting in a DCG score of 31.8.

Query	Model	DCG
“encode string with utf-8”	Ward	15.3
“Create unicode animations in a terminal”	KMeans	14.4
“create simple line plot”	Ward	18.7
“create contextual embeddings from strings”	Ward	23.4
“setup a web server”	folder	16.0
“sent email to users based on their profile”	Ward	12.2
“compress file to common formats”	Ward	9.35
“find dependency relations in pip project”	folder	10.6

Table B.2: User study queries resulting in high DCG.

B.1 Multiple Functionality Failures

Query	Functionalities	Model	DCG
“convert xslx to csv”	read xslx then write csv	KMeans	4.73
“authenticate and save oauth token google”	authenticate token, save token	Ward	4.72
“cyclic complexity code analysis for python”	cyclic complexity, code analysis, python	Ward	4.34
“read and write metadata from image files”	read image metadata, write image metadata	KMeans	2.61

Table B.3: Examples of queries that included multiple functionalities and showed low performance.

During the study it was also found that using a query with multiple functionalities resulted in less accurate results. A few examples of queries describing multiple functionalities and which achieved a DCG score lower than 5 can be found in Table B.3.

B.2 Reducing Bias

Model	Annotated Queries Raw	Annotated Queries Clean
Ward	37	26
KMeans	40	26
Folder	47	31

Table B.4: Number of annotated queries for evaluation when filtering ones only annotated by evaluated model.

In Table B.6 the NDCG scores when not ignoring not annotated repositories for the queries are presented. These values were calculated in an attempt to reduce the bias since this will have the same result as all non annotated repositories being annotated as a 0 which is the worst possible score.

Aggregation \ $NDCG_{dataset}$	Raw	Clean
Embedding	0.63	0.57
Repository Mean	0.46	0.37
Folder	0.48	0.41
KMeans	**0.57**	0.48
Ward	0.49	**0.49**

Table B.5: NDCG scores for different aggregation methods and k-values where the evaluated models own annotated queries are filtered. Baseline scores are above dotted line and best non-baseline scores are in bold font.

Aggregation \ $NDCG_{dataset}$	Raw	Clean
Embedding	0.61	0.54
Repository Mean	0.37	0.30
Folder	0.39	0.33
KMeans	**0.51**	**0.52**
Ward	0.45	0.49

Table B.6: NDCG scores for different aggregation methods and k-values where the NDCG algorithm counts non-annotated results as worst possible matches. Baseline scores are above dotted line and best non-baseline scores are in bold font.

For easier comparison between tables 5.4, B.6 and B.5 we created Table B.7. From these values it looks like the scores in Table 5.4 may exaggerate the effectiveness of the methods, but the general comparative scores between models remain and thus the dataset should only minimally be biased towards specific aggregations.

Aggregation \ Table	Raw			Clean		
	5.4	B.5	B.6	5.4	B.5	B.6
Embedding	0.63	0.63	0.61	0.57	0.57	0.54
Repository Mean	0.46	0.46	0.37	0.37	0.37	0.30
Folder	0.48	0.48	0.39	0.41	0.41	0.33
KMeans	0.55	0.57	0.51	0.52	0.48	0.52
Ward	0.55	0.49	0.45	0.49	0.49	0.49

Table B.7: Comparison between NDCG values from tables 5.4, B.6 and B.5.

INSTITUTIONEN FÖR DATAVETENSKAP | LUNDS TEKNISKA HÖGSKOLA | PRESENTERAD 2023-02-28

EXAMENSARBETE Recommending Relevant Open Source Software using Semantic Functionality Search
STUDENTER Filip Hedén, Nils Barr Zeilon
HANDLEDARE Christoph Reichenbach (LTH), Emil Wåreus (Debricked AB)
EXAMINATOR Niklas Fors (LTH)

Rekommendation av relevant programvara med hjälp av semantisk funktionalitetssök

POPULÄRVETENSKAPLIG SAMMANFATTNING **Filip Hedén, Nils Barr Zeilon**

Semantisk funktionalitetssökning har redan visat framgång vid rekommendation av relevanta kodstycken som mjukvaruingenjörer kan återanvända. Med detta arbete introducerar vi kombinationer av vektorer för att utvidga användningsområdet till rekommendationer av samlingar av öppen källkod eller mjukvarukomponenter.

Som mjukvaruutvecklare förväntas vi inte längre bara skriva kod, utan också återanvända och bygga vidare på andra utvecklares programvara. Återanvändandet av kod har blivit möjlig tack vare fritt tillgänglig kod som publiceras på internet, så kallad *öppen källkod*. Öppen källkod distribueras ofta i form av ett "repository", eller arkiv, en samling filer med kod som beskriver programvaran. Med tillväxten av öppen källkod som är tillgänglig vill vi enkelt hitta relevant programvara för våra specifika behov, ofta genom en sökmotor.

I vårt examensarbete har vi utvecklat en sådan sökmotor med hjälp av *semantisk funktionssökning*. Semantisk funktionalitetssökning fångar viktig semantik, eller betydelse, i kod och samlar informationen i numeriska vektorer, eller listor av siffror. Användare av vår sökmotor kan sedan söka efter någon funktionalitet eller ett användingsområde, och vårt system kan rekommendera det bäst passande arkivet till användarens behov.

Vårt arbete jämför rekommendation av arkiv genom användning av enskilda vektorer mot användningen av vad vi kallar för arkiv-vektorer, vilka är kombinationer av grupper av liknande vektorer i ett arkiv och tillsammans representerar viss kodfunktionalitet. Vi utvärderade vår metod med en användarstudie där erfarna programmerare kunde rangordna relevansen av de arkiv som vår sökmotor rekommenderade.

Från den användarstudien kan vi konstatera att de mest relevanta rekommendationerna uppnåddes genom att rekommendera de bäst matchande arkiv med en kombination av kod-vektorer och kombinationer av kod-vektorer.

Genom att endast använda kombinationer av vektorer så rekommenderades arkiv med något mindre relevanta rekommendationer. Men då sökutrymmet blir mindre när kombinerade vektorer används förbättras också skalbarheten för rekommendationssystemet. Detta på grund av att systemet inte behöver jämföra förfrågan mot ett lika stort antal vektorer för att kunna avgöra den bästa matchen.

Sammanfattningsvis visar våra resultat att genom att använda kombinationer av vektorer kan vi antingen öka relevansen för rekommenderade arkiv eller öka skalbarheten för sökmotorn.