Active Code Search: Incorporating User Feedback to Improve Code Search Relevance

Shaowei Wang, David Lo, and Lingxiao Jiang
School of Information Systems, Singapore Management University
{shaoweiwang.2010,davidlo,lxjiang}@smu.edu.sg

ABSTRACT

Code search techniques return relevant code fragments given a user query. They typically work in a passive mode: given a user query, a *static* list of code fragments sorted by the relevance scores decided by a code search technique is returned to the user. A user will go through the sorted list of returned code fragments from top to bottom. As the user checks each code fragment one by one, he or she will naturally form an opinion about the true relevance of the code fragment. In an active model, those opinions will be taken as feedbacks to the search engine for refining result lists.

In this work, we incorporate users' opinion on the results from a code search engine to refine result lists: as a user forms an opinion about one result, our technique takes this opinion as feedback and leverages it to re-order the results to make truly relevant results appear earlier in the list. The refinement results can also be cached to potentially improve future code search tasks. We have built our active refinement technique on top of a state-of-the-art code search engine—Portfolio. Our technique improves Portfolio in terms of Normalized Discounted Cumulative Gain (NDCG) by more than 11.3%, from 0.738 to 0.821.

Categories and Subject Descriptors: D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement

General Terms: Algorithms; Experimentation

Keywords: Code Search; User Feedback; Active Learning

1. INTRODUCTION

Millions of open source and industrial software systems have been developed and deployed. Maintaining these systems requires constant searching through various code bases and documents and relate different parts of the code together. Also, the development of new systems can benefit from reusable knowledge hidden in many existing systems if the developers can search through existing code and find relevant code for reuse. Studies in the literature have proposed various code search techniques that can return pieces of code considered to be relevant to a user's query [1, 8].

One factor that potentially affects the effectiveness of code search is the difficulty in formulating a precise query that matches the relevant code fragments needed by users. Often, users could not exactly specify what they want in the form of a set of keywords or constraints, but they are often able to decide whether a given code fragment satisfies their needs. Given a list of code fragments returned by a code search engine, some code fragments might be close to what a user wants, while others might be very different. As a user checks each code fragment one at a time, he or she will naturally form an opinion about its relevance. It will be beneficial to utilize user opinions on code fragments that he or she has checked to improve the results.

This paper aims to improve code search relevance by integrating users' feedback. Our approach records users' opinions about each code fragment when they check through a list of results returned by a search engine and refines the list so that code fragments that are more relevant to users' need would appear earlier in the list, which save users' time. It can be built on top of any code search engine that takes textual keywords or descriptions as query inputs. The feedback required by our approach does not impose much additional effort on users, as they would need to navigate search results anyway to decide whether the results match their needs; the only difference is that they need to *explicitly* indicate their opinions about a search result when using our approach.

Specifically in this paper, we build our approach on one of the state-of-the-art code search engines, Portfolio [8], to incorporate user feedback. Users of our approach first receive a list of search results from Portfolio for their query as usual. Then, they can provide relevance feedback on each search result when they check through the list one by one. The feedback is expressed as a label in a 4-point Likert scale, where 1,2,3, and 4 indicate whether the result is completely irrelevant, mostly irrelevant, mostly relevant, and highly relevant. Once the relevance feedback for a search result is recorded, our approach incorporates this feedback to re-sort the remaining search results. This is performed based on the semantic and structural similarity between each of the remaining results and all known relevant or irrelevant results. The goal is to have search results that are more likely to be relevant to be re-arranged nearer to the top of the list. Our approach can also store past feedbacks collected from previous queries to potentially improve both the current and future code queries. We refer to this as *active code search*—

ASE'14, September 15-19, 2014, Vasteras, Sweden.

http://dx.doi.org/10.1145/2642937.2642947.

users can more actively affect the output of a code search engine by providing a series of relevance feedback.

In the information retrieval community, Rocchio is a popular way to incorporate relevance feedback by refining a query based on a set of labeled results [7]. Our approach is a customized reference feedback mechanism tailored for code search. Different from Rocchio, which only considers textual information, we also consider structural information extracted from source code.

We have evaluated our approach with 70 queries on a code base containing 19,414 programs written in C and/or C++, comprised of about 169 million lines of code. Our evaluation shows that Portfolio achieves an average Normalized Discounted Cumulative Gain (NDCG [5]) score of 0.738 for the 70 queries on the code base, while our active code search approach (Portfolio^{active}) achieves 0.821. We have also compared our approach against Portfolio merged with Rocchio (Portfolio^{rocchio}) and find that Portfolio^{rocchio} achieves a lower average NDCG score of 0.764.

The structure of this paper is as follows. We present our active code search approach in Section 2 and zoom in to the refinement engine component in Section 3. Our empirical evaluation results are presented in Section 4. We discuss related studies in Section 5. We finally conclude and mention future work in Section 6.

2. APPROACH OVERVIEW

Our active code search framework is shown in Figure 1. It consists of two major processing components (shown as rectangular blocks in the figure): Code Search Engine and Refinement Engine. The interaction between users and these two major components would result in a list of results (in the rounded rectangular block) containing potentially relevant code fragments. As a user can provide feedback piece by piece incrementally, the list of search results would be refined in multiple iterations, as illustrated by the circular arrows. The sequence of the interactions in our active code search framework is described in the following paragraphs.

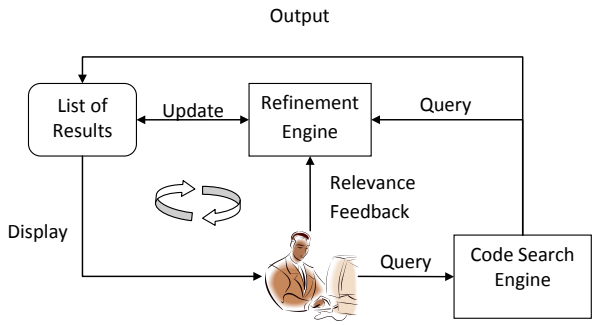

Figure 1: Active Code Search: Overall Structure

First, a user posts a query to a passive code search engine. The search engine takes the user query and returns a list of code fragments (i.e., the "List of Results" block in Figure 1) sorted according to the scores of the code fragments calculated internally by the search engine. The earlier a code fragment appears in the list, the more similar the search engine thinks it is to the user query. In this paper, we use Portfolio [8] in the Code Search Engine block.

Second, as the user navigates through the list of results one by one, he or she can provide relevance feedback. For example, after the user investigates the result at the top of the list, the user forms a judgement if the result is relevant. The judgement can be expressed as a label for the result in a 4-point Likert scale, where 1, 2, 3, and 4 indicate whether the result is completely irrelevant, mostly irrelevant, mostly relevant, and highly relevant, respectively. Each judgement for the results investigated by the user is used as input to the Refinement Engine block one at a time.

Third, as the refinement engine receives a piece of relevance feedback, it refines the ordering of the results in the list that have not been investigated by the user, aiming to improve the overall relevance of the refined list. For example, an uninvestigated code fragment that is originally at the 5th position could be shifted to the 48th position, as it is similar to some investigated code fragments that are given low relevance scores; on the contrary, an uninvestigated code fragment at the 48th position could be shifted to the 4th position, as it is similar to some investigated code fragments receiving high relevance scores. The process that involves refining a list of results, displaying a refined list, and providing relevance feedback repeats until the user decides to stop or when the list is exhausted. When a user stops, it could correspond to cases where the user has found what he or she wants, or where the user decides to accept something marginally relevant and use the returned code from there, or where the user gives up and decides to complete his or her task without reference code, etc. In the refinement engine, the feedbacks of previous queries and their corresponding refined results are cached. When a new query is posted, if a similar query is identified in the cache, the refined results of the similar query are returned. We describe how we identify the relevant refined results in Section 3.

Our active code search approach does not query the passive code search engine during the refinement, and thus does not need changes to the passive code search engine and can be easily integrated with any engine that takes user queries in the form of textual keywords or descriptions.

3. REFINEMENT ENGINE

This section describes the core component of our approach in detail. Figure 2 illustrates the structure of our refinement engine. The engine takes in the original user query, a list of results, the relevance feedback from users expressed in 4-point Likert scores, and outputs a refined list of results that shows potentially more relevant results nearer to the top.

Our refinement engine has several data blocks (in rounded rectangles): "Results", "Labeled Results", "Unlabeled Results", "Refined Query Representation", and "Reordered Results". Results are the list of results displayed to users so far. Labeled results are results that have received users' relevance feedback. Unlabeled results are those that have not received relevance feedback from users. Refined query representation is used internally in our refinement engine to represent the combined effect of the original query and the results that have been investigated and labeled by the user. Reordered results are the refined list outputted by the refinement engine after taking user feedback into consideration and will be presented to users for additional feedback.

Our refinement engine has also several processing blocks (in rectangles): "Update Query Representation", "Parameter Tuning","Reorder Results", and "Cache Processor". These processing blocks work together to refine the list of results for the original user query based on the relevance feedback.

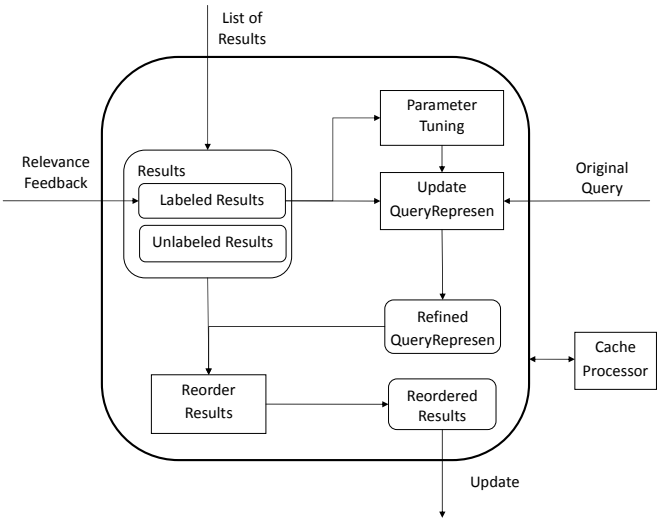

Figure 2: Refinement Engine Component

The "Update Query Representation" block produces the refined query representation from labeled results and the original query. This refined query representation is used by the "Reorder Results" block to produce the refined results. The "Update Query Representation" block accepts parameters that are to be tuned periodically as more relevance feedback is received. This is done by the "Parameter Tuning" block. The "Cache Processor" block stores the original queries that the refinement engine have processed before and their corresponding labeled results, refined query representations, and reordered results. This block also checks if a new input query closely matches a past query; if it does, it will bootstrap the refinement engine with the cached data. We elaborate these processing blocks in Sections 3.1, 3.2, 3.3, and 3.4.

3.1 Update Query Representation

In this block, we incorporate information from search results that have received relevance feedback (i.e., Labeled Results) into the original query. We convert the original query into its representative vectors. We also convert search results that have received relevance feedback into their representative vectors. We then update the representative vectors of the original query to a refined query representation. To elaborate "Update Query Representation", we present some definitions first, and then the algorithm.

3.1.1 Query and Result Representations

We first introduce the representations for a query and a result that can be used to transform a textual query or a code fragment into vectors of numerical scores. We consider two representations: *semantic* and *structural*.

Definition 1 (Semantic Representation). *In the semantic representation, a query and a code fragment are viewed as a bag of words. We use standard tokenization, stop word removal, identifier splitting, and stemming to convert a query or a code fragment into a bag of words [7]. The semantic score of a word is given by the product of its* term frequency *and* inverse document frequency *$(tf * idf)$ [7]. The term frequency (tf) of a word is the number of times the word appears in the query or code fragment normalized by the total number of words in the query or code fragment. The inverse document frequency (idf) of a word is the logarithm of the total number of documents (i.e, the number of methods in the code base in our code search setting) divided by the number of documents that contain the word. Given a query or code fragment q, we denote the vector of semantic scores representation of q as $VScore(q)_{sem}$.*

Example. Let a query q be *lock unlock file*, then the term frequency of words *lock*, *unlock* and *file* are 0.33, 0.33, 0.33 respectively. Also, let the words *lock*, *unlock* and *file* appear in 270,872, 154,029, and 800,672 methods and there are totally 7,916,458 methods in the code base, so the inverse document frequencies for the words calculated by the equation $idf(w, D) = log \frac{|D|}{|\{d \in D : w \in d\}|}$ are 1.46, 1.71, and 0.99, respectively, where w is a word in the query and d is a method in the code base D. Finally, the semantic scores are 0.48, 0.54, and 0.33 for *lock*, *unlock* and *file*, respectively.

Definition 2 (Structural Representation). *In the structural representation, a query and a code fragment are viewed as a bag of function calls. The structural score of a function is given by the product of its term frequency and inverse document frequency. Given a query or a code fragment r in the search results, the term frequency of a function in r is the frequency of the function being called in r normalized by the total number of function calls in r; the inverse document frequency of a function is the logarithm of the total number of code fragments in the search results divided by the number of code fragments in the search results that call the function. We denote the vector of structural scores of r as $VScore(r)_{str}$. Note that the original query entered by a user does not contain method calls, thus it is represented by a vector of zeroes. However, as we incorporate results that have received relevance feedback from users to the original query, the query's vector of structural scores would be updated.*

Example. Consider a code fragment r which is a function *ScLockedFile*. There is only one function *unlock* called in *ScLockedFile*. So the term frequency of *unlock* is 1. There are totally 50 search results returned, and the function *unlock* is called by 2 code fragments in the results; so its inverse document frequency value is $log \frac{50}{2}$=1.40. Finally, the $VScore(r)_{str}$ for *unlock* is 1.40.

3.1.2 Vector Operations

We also define operations that are applied to vectors and help to make it easier to describe the algorithm in the following subsections of this paper.

Definition 3 (Vector Summation and Division). – *Let $v[i]$ be the score in a vector v corresponding to a word or a function i. In the case that the word or the function i does not have a corresponding entry in v, let $v[i]$ returns 0. Given two vectors $v1$ and $v2$, the* summation *of these two vectors would result in a new vector vr, and $\forall i : vr[i] = v1[i] + v2[i]$. Consider a vector v and a constant c, the division of vector v by c would result in a new vector vr, and $\forall i : vr[i] = v[i]/c$.*

We also need a similarity metric among vectors so that we can gauge the refinement of queries and search results. In this paper, we measure vector similarity by using the well-known cosine similarity [7].

3.1.3 Algorithm

The procedure for "Update Query Representation" is shown in Algorithm 1. The procedure takes in a set of search results labeled so far (LBL), the new feedback ($fback$), the set of unlabeled results ($ULBL$), the original user query $origquery$, and a set of weights ($\alpha_1, \alpha_2, \alpha_3, \alpha_4$) that determine the contributions of labeled results with the Likert scores 1, 2, 3, and 4 respectively. Weight $\alpha_1, \alpha_2, \alpha_3, \alpha_4$ are set to be -0.3, -0.1, 0.1, and 0.5 initially. The procedure updates the set of labeled results LBL and unlabeled results $ULBL$, and creates a refined query representation $refquery$ which consists of two vectors $refquery_{sem}$ and $refquery_{str}$ which are the semantic and structural representations.

To transform the inputs to the outputs, the procedure works in the following steps. We split the results in LBL into four sets based on the relevance scores (Line 11). After this step, we compute the semantic *center* of each set (Line 12) by the following equation:

$$Ci_{sem} = \frac{\sum_{r \in LBL_i} VScore_{sem}(r)}{|LBL_i|}$$

Similarly, we compute the structural *center* of each set (Line 13) by the following equation:

$$Ci_{str} = \frac{\sum_{r \in LBL_i} VScore_{str}(r)}{|LBL_i|}$$

The (semantic or structural) center of each set is a vector that is the summation of the (semantic or structural) vectors of all results appearing in the set normalized by the size of the set (i.e., the number of results in the set). We then compute the semantic refined query representation ($refquery_{sem}$) by combining the centers of the corresponding four sets (Ci_{sem}) with the semantic score vector of the original query ($origquery_{sem}$) (Line 14). The structural refined query representation ($refquery_{str}$) is computed in a similar way (Line 15). This refined query representation ($refquery_{sem}$,$refquery_{str}$) is then used to help reorder the search results in Section 3.2.

3.2 Reorder Results Block

Algorithm 1 Update Query Algorithm

1: **Procedure UpdateQuery**
2: **Input:**
3: LBL: Labeled search results
4: $fback$: New feedback, i.e., a new Likert score to an unlabeled search result
5: $ULBL$: Unlabeled search results
6: $origquery$: Original user query
7: $\alpha_1, \alpha_2, \alpha_3, \alpha_4$: The weights of contributions of labeled results with the Likert score 1, 2, 3, and 4 respectively
8: **Output:** Updated LBL, $ULBL$, and a refined query representation $refquery$
9: **Method:**
10: Add $fback$ into LBL, and remove the result labeled by $fback$ from $ULBL$
11: Let LBL^1, LBL^2, LBL^3, and LBL^4 be the sets of results in LBL with Likert scores 1, 2, 3, and 4 respectively
12: Compute the semantic centers of LBL^1, LBL^2, LBL^3, and LBL^4 and denote them as $C1_{sem}$, $C2_{sem}$, $C3_{sem}$, and $C4_{sem}$
13: Compute the structural centers of LBL^1, LBL^2, LBL^3, and LBL^4 and denote them as $C1_{str}$, $C2_{str}$, $C3_{str}$, and $C4_{str}$
14: Let $refquery_{sem} = VScore_{sem}(origquery) + \sum_{i=1...4} \alpha_i \times Ci_{sem}$
15: Let $refquery_{str} = VScore_{str}(origquery) + \sum_{i=1...4} \alpha_i \times Ci_{str}$
16: Return LBL, $ULBL$, and ($refquery_{sem}$, $refquery_{str}$)

In this block, we sort the unlabeled results based on their similarity with the refined query representation. The pseudocode is shown in Algorithm 2. It takes in the refined query representation $refquery$ generated by Algorithm 1 and a list of unlabeled search results $ULBL$. It outputs a reordered $ULBL$ in the following steps. First, it iterates through the list of unlabeled results to compute the *structural* and *semantic similarity scores* between the score vectors of each result and those of $refquery$, i.e., $refquery_{sem}$ and $refquery_{str}$ (Lines 8–9). It then takes the average of these two scores as the *overall similarity score* for each unlabeled result (Line 10). Then, it reorders the unlabeled results according to their overall similarity scores in the descending order (Line 12).

Algorithm 2 Reorder Result Algorithm

1: **Procedure ReorderResults**
2: **Input:**
3: $refquery$: Refined query representation
4: $ULBL$: Unlabeled results
5: **Output:** Reordered $ULBL$
6: **Method:**
7: **for all** r in $ULBL$ **do**
8: Let $sim_{sem} = cos(VScore(r)_{sem}, refquery_{sem})$
9: Let $sim_{str} = cos(VScore(r)_{str}, refquery_{str})$
10: Let $sim_{overall} = \frac{sim_{sem}+sim_{str}}{2}$
11: **end for**
12: Sort all results in $ULBL$ in the descending order according to their overall similarities to $refquery$ as computed at Lines 8–10
13: Return $ULBL$

3.3 Parameter Tuning

Our refinement engine takes in 4 weight parameters: α_1, α_2, α_3 and α_4. These 4 parameters are initially set to take the following values: -0.3, -0.1, 0.1, and 0.5, respectively. We re-tune these weight parameters whenever we receive a new relevance feedback rating a result with Likert score 3 or 4 (i.e., the result is mostly or highly relevant). Our algorithm tunes the parameters by trying many possible parameter settings one at a time. For each parameter setting, we consider its effectiveness on the set of labeled data known so far. We pick the parameter setting which is the most effective (i.e., it achieves the highest NDCG on the set of labeled data known so far).

The pseudo-code is shown in Algorithm 3. Let us define a notation $AP(a, b, s)$ to represent an arithmetic progression (AP) between a and b with a step s. We initialize two ranges α_{1-2} and α_{3-4} to be $AP(-0.5, 0.4, 0.1)$ and $AP(0, 0.9, 0.1)$ respectively (Line 6). The first is the range of possible parameter values for α_1 and α_2. The second is the range of possible parameter values for α_3 and α_4.

Then we try to adjust the 4 parameters by trying different combinations of values picked from the sets α_{1-2} and α_{3-4} (Line 8). After a combination of parameters are picked, we evaluate its effectiveness on the set of labeled results known so far LBL (Line 9). We simply reorder LBL using the parameter combination setting and compute NDCG. The larger the resultant NDCG score is, we assume the better a parameter combination is. We repeat the two steps (i.e., Line 8 and 9) until we find a local optimum (Line 10). We detect a locally optimal setting for $\alpha_1, \alpha_2, \alpha_3, \alpha_4$ by trying all combinations of values picked from the sets α_{1-2} and α_{3-4} in order and looking for a combination whose resultant NDCG score is 1 or no smaller than the NDCG scores of its neighboring configurations in the combinations of parameter

values in the arithmetic progressions. We finally output the local optimal setting of the 4 parameters (Line 11).

Algorithm 3 Parameter Tuning Procedure

```
1: Procedure ParameterTuning
2: Input:
3: LBL: Labeled results
4: Output: Parameters α1, α2, α3, α4
5: Method:
6: Initialize two sets α1−2 and α3−4 to be AP(−0.5, 0.4, 0.1) and
   AP(0, 0.9, 0.1) repectively
7: repeat
8:    Adjust the value of the 4 parameters
9:    Evaluate effectiveness of the adjusted parameters on LBL
10: until a local optimum is reached
11: Output the local optimum
```

3.4 Cache Processor

Once a user posts a new query, the "Cache Processor" block checks whether there exists a highly similar query in the cache. If a highly similar query is identified, its corresponding cached results will be used to bootstrap the refinement engine. To identify whether a highly similar query exists, this block computes the similarity of the new query with each of the old queries. The similarity of a new query q_{new} and an old query q_{old} is computed by taking the cosine similarity of their corresponding semantic vectors:

$$Similarity_{sem} = cos(VScore(q_{new})_{sem}, VScore(q_{old})_{sem})$$

We rank the old queries based on their similarity scores. The top-one old query whose similarity is larger than a threshold t is identified as the highly similar query (ties are randomly broken). If no old query has similarity above t, then no highly similar query is identified. In this study, by default, we set t to 1 which means only the exactly same query will be identified.

4. EXPERIMENTS

4.1 Experimental Settings

The code base we use in this work is from FreeBSD[1]. We download 49,889 program versions from this code base, but use only the latest version for programs that have multiple versions. Finally, we use 19,414 programs written in C and/or C++ in our study. The total size of these programs is around 36GB, and they contain about 169 million lines of code, 8 million functions, and 2 million files. We have used 70 queries created by Portfolio's author [8]. All these queries are formulated as set of keywords to address some programming tasks reported in Portfolio's user study.

We involve users for evaluation and use a simple web application written in PHP to display search results and collect user feedback. In the user study, we have 10 participants, 9 of them are PhD students who have at least of two years of Java and C++ programming experience, and the other is a professional software engineer who has three years of Java and C++ programming experience. Each participant is assigned a number of queries[2] and asked to examine the top fifty search results for each query and provide a relevance score in a 4-point Likert scale for every result.

[1] `ftp://ftp.freebsd.org/pub/FreeBSD/distfiles/`

[2] Each query is assigned to only one participant.

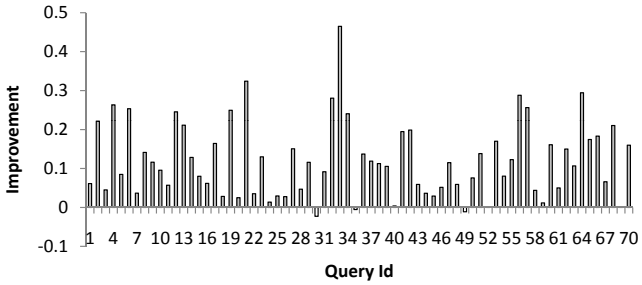

Figure 3: Improvement of Portfolio^{active} over Portfolio^{original} in terms of NDCG

Table 1: Three sample cases where Portfolio^{active} improves the NDCG of the search results the most

Query	NDCG of Portfolio^{original}	NDCG of Portfolio^{active}	Reordering samples
mscdex emulation	0.557	0.738	14→2, 15→3
filename lock controlled access	0.596	0.873	46→2, 47→3
digital camera to jpeg	0.612	0.792	30→2, 50→8

4.2 Evaluation Results

We build our approach on top of Portfolio and thus we need to compare the two approaches. We compare the NDCG score of Portfolio^{active} with that of the original Portfolio (Portfolio^{original}). Our evaluation shows that the original Portfolio on average achieves an NDCG score of 0.738 while Portfolio^{active} on average achieves an NDCG score of 0.821, a 11.3% improvement. Figure 3 presents a detailed comparison of the NDCG score for every query. Portfolio^{active} wins in 67 cases while marginally performing worse for 3 other cases (query number 35, 49, and 52). We also perform a Wilcoxon signed-rank test on the NDCGs and find that the improvement achieved by Portfolio^{active} is statistically significant (p-value < 0.05).

For illustration, Table 1 shows the top three cases where Portfolio^{active} improves the NDCG of the search results the most. For example, for "mscdex emulation", Portfolio^{active} reorders two highly relevant results from positions 14 and 15 to positions 2 and 3 just after one feedback, and improves the NDCG from 0.557 to 0.738.

In the default setting, for each code search task, users give a feedback to each of the first 50 results and after each feedback we apply our refinement engine. We would like to test the effectiveness of our active code search approach with different numbers of feedbacks from users (denoted as K_f). In this experiment, users only give feedback to the first K_f results. We vary K_f in the set {1,5,10,20,30,40,50}. Table 2 compares the effectiveness of Portfolio^{original} with our Portfolio^{active} with different number of feedbacks (K_f) to refine the results. When only one feedback is given by users for refining the results, Portfolio^{active} achieves a 7.46% improvement over Portfolio^{original}. As the K_f value increases, Portfolio^{active} achieves more and more improvement until K_f reaches 30. The effectiveness of Portfolio^{active} remains constant when K_f is increased from 30 to 50.

We also want to compare the effectiveness of our approach against Portfolio with Rocchio (Portfolio^{rocchio}). Different from our proposed approach, standard Rocchio does not consider structural scores and use a static set of weights to incorporate labeled results to refine a query. In our experi-

Table 2: Comparison of Portfolio^{original} and Portfolio^{active} in terms of NDCG for different K_f

K_f	Portfolio^{original}	Portfolio^{active}	Improvement
1	0.738	0.794	7.46%
5	0.738	0.809	9.59%
10	0.738	0.816	10.55%
20	0.738	0.820	11.05%
30	0.738	0.821	11.12%
40	0.738	0.821	11.12%
50	0.738	0.821	11.12%

ments, we set the weights of Rocchio to their recommended values [7]: $a = 1$, $b = 0.75$, and $c = 0$, where a, b, and c are the weights of the original query, results labeled as relevant, and results labeled as irrelevant respectively.

We compare Portfolio^{active} with Portfolio^{rocchio} in terms of NDCG. Portfolio^{rocchio} achieves an average NDCG score of 0.764. Porfolio^{active} achieves an average NDCG score of 0.821 which is a 7.5% improvement over Portfolio^{rocchio}'s result. We also perform a Wilcoxon signed-rank test and find that the improvement achieved by Portfolio^{active} is statistically significant (p-value < 0.05).

4.3 Threats to Validity

Threats to internal validity include experimenter biases. There might be subjectivity in the relevance scores that a participant assigns to returned code fragments. Threats to external validity relate to the generalizability of our findings. We have only investigated 70 queries and considered a code base consisting of 19,414 projects and 169 million lines of code. We have also only considered the case where the value of threshold t of the Cache Processor block is set to 1. Threats to construct validity refer to the suitability of our evaluation metrics. We use NDCG, a common metric used to investigate the quality of web search engines [5, 7].

5. RELATED WORK

There are many code search approaches proposed to help users find relevant code. Some of them take textual information as input to search code. McMillan et al. propose Portfolio that takes natural language descriptions as input and outputs a list of functions or code fragments along with corresponding call graphs [8, 9]. Chan et al. propose an approach to help developers find usages of API methods given simple text phrases [1]. They use an efficient graph search algorithm to return an optimum connected subgraph that matches a query. Haiduc et al. propose a code search tool named Refoqus that is able to predict the quality of a textual query for further query reformulation [3]. Different from their work, we integrate the structural information of source code itself with text information to refine code search results. There are also some code search engines that allow users to query by specifying structural constraints [10, 11].

Gay et al. [2] use the Rocchio algorithm to incorporate user feedback to improve the performance of concern localization. They are interested in finding methods that need to be fixed given a defect report. In their approach, each method and defect report is treated as a simple textual document. Different from their work, we address a different problem (we recommend code fragments from a short textual query) and propose a new and specialized query refinement algorithm (instead of using Rocchio) which incorporates structural information and employs parameter tuning.

There are also existing studies in software engineering that employ active learning. Lucia et al. propose an approach that adds an active learning layer on top of existing clone-based bug detection tools to increase true positive rate [6]. Hayes et al. use the Rocchio algorithm to improve the quality of requirement tracing techniques which infer links between two textual documents, e.g., high level to low level requirements [4].

6. CONCLUSION AND FUTURE WORK

In this paper we propose active code search, where a user can provide feedback to code search engines and guide the engines to improve the relevance of search results. We propose a refinement engine that can take into consideration user relevance feedback: based on a set of results whose relevance feedback have been received, our engine enhances the user original query and use it to update the search results by reordering the potentially more relevant search results to the top of the list for users to see. The refinement process can be repeated for a number of times until the list is exhausted or the user decides to stop searching further down the list. Our active code search technique imposes little additional overhead on users, and can improve the relevance of search results from any passive code search engine that takes textual descriptions as user queries. We have evaluated our approach on 70 queries and find that our active code search approach on average improves the effectiveness of code search by 11.3% in terms of NDCG. We have also compared our approach against Rocchio and finds that we can improve it by 7.5%.

In this work we only apply our approach to Portfolio. We plan to apply our approach to other passive code search tools and show that the active code search paradigm can benefit those code search engines too.

7. REFERENCES

[1] W. K. Chan, H. Cheng, and D. Lo. Searching connected API subgraph via text phrases. In *FSE*, 2012.

[2] G. Gay, S. Haiduc, A. Marcus, and T. Menzies. On the use of relevance feedback in IR-based concept location. In *ICSM*, 2009.

[3] S. Haiduc, G. Bavota, A. Marcus, R. Oliveto, A. D. Lucia, and T. Menzies. Automatic query reformulations for text retrieval in software engineering. In *ICSE*, 2013.

[4] J. Hayes, A. Dekhtyar, and S. Sundaram. Advanced candidate link generation for requirements tracing: The study of methods. In *TSE*, 2006.

[5] K. Järvelin and J. Kekäläinen. Cumulated gain-based evaluation of IR techniques. *ACM Trans. Inf. Syst.*, 20(4):422–446, Oct. 2002.

[6] Lucia, D. Lo, L. Jiang, and A. Budi. Active refinement of clone anomaly reports. In *ICSE*, 2012.

[7] C. D. Manning, P. Raghavan, and H. Schütze. *Introduction to Information Retrieval*. Cambridge, 2008.

[8] C. McMillan, M. Grechanik, D. Poshyvanyk, Q. Xie, and C. Fu. Portfolio: finding relevant functions and their usage. In *ICSE*, 2011.

[9] C. McMillan, D. Poshyvanyk, M. Grechanik, Q. Xie, and C. Fu. Portfolio: Searching for relevant functions and their usages in millions of lines of code. *ACM Trans. Softw. Eng. Methodol.*, 22(4), 2013.

[10] S. Wang, D. Lo, and L. Jiang. Code search via topic-enriched dependence graph matching. In *WCRE*, 2011.

[11] X. Wang, D. Lo, J. Cheng, L. Zhang, H. Mei, and J. X. Yu. Matching dependence-related queries in the system dependence graph. In *ASE*, 2010.