CodeReco - A Semantic Java Method Recommender

by

Shashank Singh

A Thesis Presented in Partial Fulfillment
of the Requirements for the Degree
Master of Science

Approved May 2017 by the
Graduate Supervisory Committee:

I-Han Hsiao, Chair
Erin Walker
Denis Parra

ARIZONA STATE UNIVERSITY

August 2017

ABSTRACT

The increasing volume and complexity of software systems and the growing demand of programming skills calls for efficient information retrieval techniques from source code documents. Programming related information seeking is often challenging for users facing constraints in knowledge and experience. Source code documents contain multi-faceted semi-structured text, having different levels of semantic information like syntax, blueprints, interfaces, flow graphs, dependencies and design patterns. Matching user queries optimally across these levels is a major challenge for information retrieval systems. Code recommendations can help information seeking and retrieval by pro-actively sampling similar examples based on the users context. These recommendations can be beneficial in improving learning via examples or improving code quality by sampling best practices or alternative implementations.

In this thesis, an attempt is made to help programming related information seeking processes via pro-active code recommendations, and information retrieval processes by extracting structural-semantic information from source code. I present *CodeReco*, a system that recommends semantically similar Java method samples. Conventional code recommendations found in integrated development environments are primarily driven by syntactical compliance and auto-completion, whereas CodeReco is driven by similarities in use of language and structure-semantics. Methods are transformed to a vector space model and a novel metric of similarity is designed. Features in this vector space are categorized as belonging to types *signature*, *structure*, *concept* and *language* for user personalization.

Offline tests show that CodeReco recommendations cover broader programming concepts and have higher conceptual similarity with their samples. A user study was conducted where users rated Java method recommendations that helped them

complete two programming problems. 61.5% users were positive that real time method recommendations are helpful, and 50% reported this would reduce time spent in web searches. The empirical utility of CodeReco's similarity metric on those problems was compared with a purely language based similarity metric (baseline). Baseline received higher ratings from novices, arguably due to lack of structure-semantics in their samples while seeking recommendations.

DEDICATION

To my parents Anita and Hansraj Singh.

ACKNOWLEDGMENTS

I would like to thank my parents for their unconditional love and the sacrifices they make for my best interests. I am also very grateful to my advisor, Dr. Sharon Hsiao. She has always been approachable by students and inspires them to pursue research without any prejudices. Her motivation, guidance and support has helped me throughout my course of graduate studies. I am also grateful to Dr. Denis Parra and Dr. Erin Walker for spending time on reviewing my work and helping me improve it.

Thanks to members of the CSI lab, my roommates and friends for their companionship. Thank you Neha for your patience with me.

TABLE OF CONTENTS

Page

LIST OF TABLES vii

LIST OF FIGURES viii

CHAPTER

1 INTRODUCTION 1

1.1 Motivation 3

1.2 Research Questions 4

1.3 Organization 5

2 BACKGROUND AND LITERATURE 6

2.1 Recommender Systems 6

2.2 Code Similarity 6

2.3 Code Recommender Systems 9

3 CODERECO SYSTEM 11

3.1 Scope 11

3.2 Design Rationale 11

3.2.1 Similarity Metric 14

3.2.2 Preprocessing 16

3.2.3 Heuristics for Concept Extraction 16

3.2.4 Algorithm and Complexity 17

3.2.5 Configuration 17

3.3 Evaluation Methodology 18

4 USER STUDY 21

4.1 Recommendation Corpus 21

4.2 Study Design 21

CHAPTER Page

4.3 Usage Descriptive Statistics 26

5 EVALUATION 31

5.1 Evaluation Metrics 31

5.2 Evaluation Results 31

6 CONCLUSION 37

6.1 Discussions and Summary 37

6.2 Limitations and Future Work 38

REFERENCES 40

APPENDIX

A USER COMMENTS 45

B MTURK QUALIFICATION QUESTIONS 48

LIST OF TABLES

Table Page

1 CodeReco Vector Feature Categories 13

2 UserStudy Descriptive Statistics 27

3 Classroom CodeReco Weight Correlations 33

4 MTurk CodeReco Weight Correlations 33

5 Classroom Comments 46

6 MTurk Comments 47

LIST OF FIGURES

Figure Page

1 Stackoverflow - QnA 2

2 Code Intermediate Structures 8

3 CodeBroker Recommendation Example 10

4 SnipMatch Recommendation Example 10

5 Feature Extraction 14

6 CodeReco Design Architecture 15

7 User Study Workflow 22

8 Searching 24

9 Rating (Voting) 24

10 Customization 26

11 CodeReco - Exploratory Customizations 28

12 CodeReco - Rating Customizations 29

13 CodeReco Number of Queries at Word Counts 30

14 Features Feedback 35

15 Subjective User Experience 36

Chapter 1

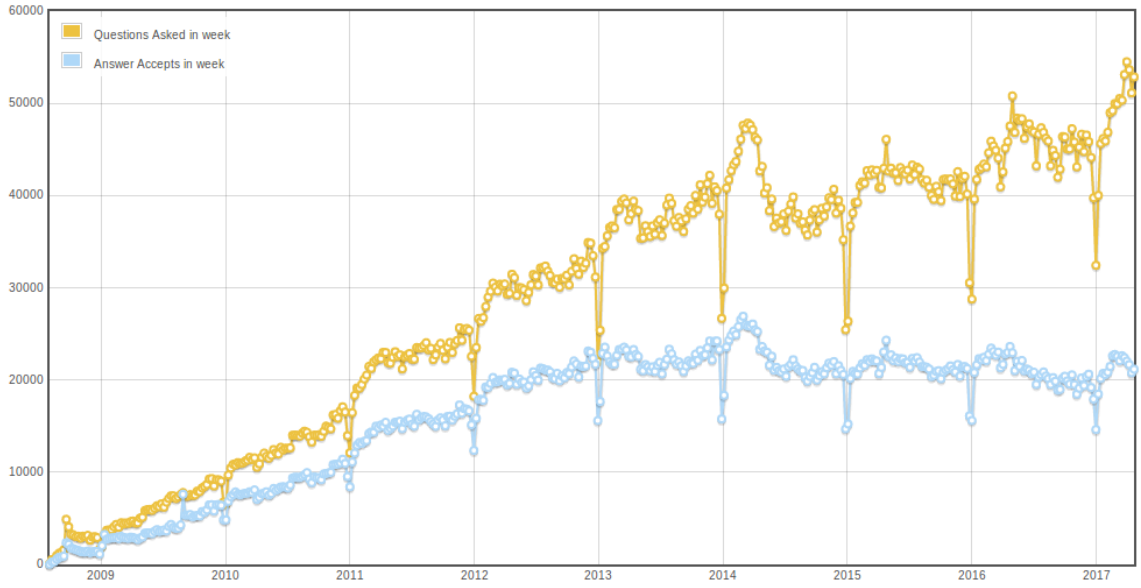
INTRODUCTION

The growing need of programming knowledge and the nature of computer programs themselves present unique challenges for programming related information seeking and retrieval. Computer programs are semi-structured text files, representing layers of information in the form of documentation, syntax, variable names and constants, blueprints (classes), callable references (methods), application programming interfaces (API's), control flows, algorithms and architecture. Such multifacated source code repositories are growing exponentially (online code hosting platform Github grew from 4+ million repositories in 2013 to 80+ million in 2017 [14]). Need of semantic information relating to source code is also growing. Community driven question and answer forums like StackOverflow[1] see a proliferating number of questions asked per week as compared to the accepted answers (figure 1).

Information seeking processes (ISP) pose cognitive challenges for users [21] like uncertainty due to lack of knowledge. Information retrieval processes (IR) on the other hand, face challenges of growing volume of heterogeneous data, retrieving user's context and semantically matching queries with corpus [1]. Improvements in information seeking and retrieval from source code is vital for achieving several objectives like searching code in open source repositories [13], encouraging reuse of software [12, 52], enhance learning by examples [32], helping developers use frameworks / API's [56] or detect plagiarism [6, 43].

[1] http://stackoverflow.com

Figure 1: Stackoverflow - QnA



http://data.stackexchange.com/stackoverflow/query/662515

Recommender systems support programming related ISP and IR by automatically recommending items relevant to the user's current context. This is particularly helpful when users do not anticipate searching or when they struggle with explicitly specifying their search query [3]. Code recommenders have already become an integral part of development environments (IDE) like eclipse [9]. However, they are predominantly based on static type checking and are limited in scope to recommending a single method call or a template for common programming constructs like loops. Research is exploring alternative uses of code recommenders like supporting novices with improved code navigation [23], api usage [18], change requests [26] and end to end collaborative software development [49]. Considering programming education, there is scope for similar applications. For example, consider that a user has forgotten to check for the termination condition in a recursive method and is facing an overflow error. The user might have faced such an error for the first time, struggling with self-regulation in

debugging [25]. He might come up with a sub-optimal web search phrase at first, that would need to be refined gradually with exploration. He eventually realizes he missed adding the recursive termination check by finding some analogous example or reading through a detailed algorithm post. If he could look at recommended methods *similar* to what he is currently working on, he could have potentially figured this out sooner.

1.1 Motivation

Need of semantic code recommendations was observed in a quizzing application for a Java programming class. The application mined web pages from Java wikibooks [19] to suggest reading articles based on the content of the question. Open source search platform Solr [2] was used to index crawled web pages and return ranked results when queried with a question content. Recommendations were reasonable for questions like *what is runtime polymorphism*, but the quality reduced when the main content of the question was a code snippet, like *what is the output of below code*. Could treating code snippets differently (more than text) improve recommendations?

The goal of this thesis is to enhance programming learning via examples [32] by providing semantic method recommendations. Based on research relating to cognitive learning via examples [36], an assumption is made that relevant semantic method recommendations can sample higher level concepts, best practices, common pitfalls and reduce learning barriers [20]. This thesis focuses on retrieving *semantic* method recommendations from a code corpus, the main objective of the CodeReco system. CodeReco extracts features via parsing the abstract syntax tree (AST) of methods, and a novel similarity metric is defined. These features are aggregated into categories

[2] http://lucene.apache.org/solr/

like *signature*, *structure*, *concepts* and *language* for incorporating user controlled personalization [31]. Two user studies were conducted to analyze the effectiveness of the novel similarity metric and gain insights on the *kind* of similarity users seek the most.

1.2 Research Questions

There are research questions relating to both the characteristics of method recommendations and their overall user utility. Would complete method recommendations (that may not be syntactically compatible) empirically help users in completing their programming tasks? Would method recommendations pro-actively reduce information seeking load from the users? Irrelevant recommendations could increase frustration among users, so how do we ensure recommendations are relevant?

One can consider a weighted keyword match approach to be a baseline indicator of relevancy. Higher the weighted keyword match, more relevant the sample. This language *baseline* is a common default option in open source search indexing engines like Solr, and a recommender system could be implemented by 'pro-actively querying' such search engines. CodeReco on the other hand, claims to be a semantic Java method recommender, but what does *semantic* method recommendation mean? Source code contains natural language (comments, naming convention, etc) and program syntax (expressions, loops, conditions, etc). From a natural language perspective, semantic information could mean synonymy, hyponymy, sense of words and their dependencies, latent topics, etc. From a programming language perspective, semantics (*structure-semantics* here onwards) could mean operational semantics (steps of execution or flow-chart), denotational semantics (function from domain input to domain output),

axiomatic semantics (contracts or api), etc. Conventional search and recommendation systems either treat code as entirely a natural language document (keyword based) or as an axiomatic representation (compatible API's and methods). CodeReco attempts to recommend method samples by extracting both natural language and programming language features, and treats them separately in a novel similarity metric measuring semantic similarity. Would this similarity metric outperform the language baseline in sampling conceptually similar examples?

This thesis evaluates below hypotheses:

H1 - Distinguished usage of language and structure-semantic features will yield conceptually similar examples

H2 - Method recommendations help users complete their programming tasks

H3 - Method recommendations reduce conventional search time while programming

H4 - Treating methods as a combination of language and structure-semantic features will yield more helpful recommendations for users as compared to a pure language treatment

1.3 Organization

This introduction is followed by literature and background (Chapter 2) to provide sufficient context to the reader. The design rationale, algorithm design and interface of CodeReco are presented next(Chapter 3), along with our methodology for evaluation. The user study is described in detail next (Chapter 4), followed by evaluation results (Chapter 5) and conclusion (Chapter 6) .

Chapter 2

BACKGROUND AND LITERATURE

2.1 Recommender Systems

Recommender systems consist of software tools and techniques that suggest items (news, articles, movies, etc) to users. There have several applications, most common ones being increasing sales and user satisfaction [37]. As a high level overview to a vast area of active research, these systems suggest items by predicting which item(s), from a possible set of items, would be most relevant or useful to the user. For predicting user utility of a new item , they use techniques like finding similar items based on user's usage history (content based), similar users based on usage history and items brought by them but not by current user (collaborative), knowledge acquired like usage patterns and associations (knowledge-based) or a combination of these techniques (hybrid). Some recommender systems are offline (asynchronous) while others could be dynamic (real time recommendations). Recommender systems also have novel applications in technology enhanced learning [28]. One of the most important characteristics of recommender systems are their notions of similarity between items or users.

2.2 Code Similarity

Significant research has been done to find similarities in source code utilizing not just the text of the source document but also intermediate transformations like abstract syntax trees (AST), bytecode and binary executables. An extensive summary of all

such techniques has been noted by Ragkhitwetsagul, Krinke, and Clark [35]. Since code files are analogous to text files, research in code similarity closely evolved with research in text similarity. As background for this thesis, I focus on applications of code similarity in areas of plagiarism detection, software development and education.

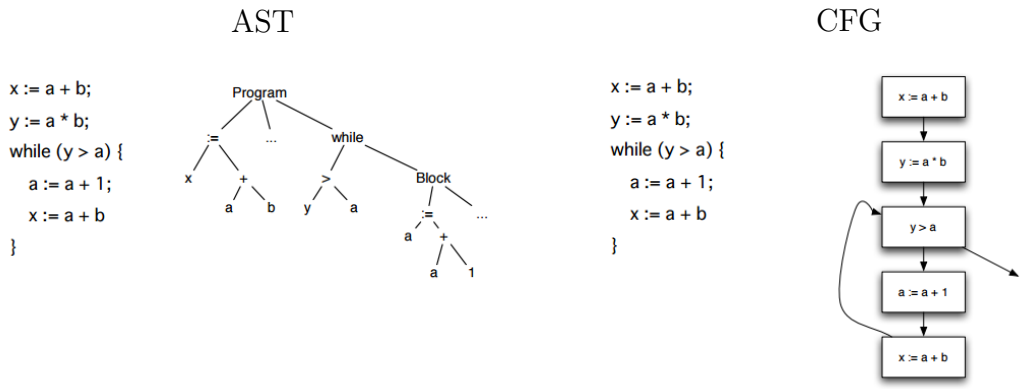
Code similarity was most extensively studied from a plagiarism detection perspective. Initial approaches involved treating code as a sequence of tokens, as done in MOSS [43], a plagiarism detection system still used in academia[3]. Documents are split into chunks either by words or sentences. 1 to n gram sequences of these chunks are hashed. Subsets of hashes are selected as fingerprints. Higher the partial matches in fingerprints between two samples, higher their similarity and more their likelihood of being copies. Bowyer and Hall [5] report that for 75 to 120 mid-sized programs ($\approx$ 200 lines) queried in MOSS, results were available in a day.

The conventional approach for search oriented information retrieval has been treating code as a text document. Documents are transformed into frequency weighted vector representations (tf-idf) where the dimensions of these vectors are words in the vocabulary [39]. Vector similarity metrics like cosine similarity are used. Vector space models, being relatively simple algebraic models, are highly scalable and extensively used in search engines like lucene[4]. Extensions of vector space models like latent semantic indexing [7] is used to detect semantic *topics* within code repositories that can be further used for clustering [22]. However, relying on the simplicity of term frequencies suffers from the major drawback of loosing all sequential or structural information.

[3]https://theory.stanford.edu/~aiken/moss/

[4]https://lucene.apache.org/core/2_9_4/scoring.html

Figure 2: Code intermediate structures



https://www.cs.umd.edu/class/fall2010/cmsc631/data-flow.pdf

To increase precision in finding structural overlaps, code AST's (figure 2) can be compared with tree edit distance metrics[2]. Higher the distance, lesser the similarity. Rivers and Koedinger[38] use this approach in their intelligent tutoring system for providing hints to students for editing their methods to match a solution in as few edits as possible. They were able to successfully generate a hint-chain to final solutions 98.3% of the time. Another approach similar to AST's are graph based that involve generating control flow graphs (figure 2) or dependency graphs[24] from code, and then calculating graph edit distances. Graph approaches enable highly semantic representations (analogous to flowcharts). But both AST and graph approaches are hard to scale since comparing graphs and tree structures is computationally expensive [55, 54].

Sridhara et al. [45] worked towards automated summary generation for methods to improve program comprehension and maintainability. They did so by predicting most important statements within the method. Wong, Liu, and Tan [51] also worked on summary generation, but with a different approach. They first detect clones using structural similarity, and from these clones select relevant documentation to apply to

the sample. Such distinct treatment of structure and language is also used in existing recommenders like CodeBroker [53].

2.3 Code Recommender Systems

Code recommender systems are tools that take in a code snippet and return code recommendations. An extensive survey of code recommender systems is reported by Mens and Lozano [29]. Amongst them, those similar to CodeReco are Strathcona [16], CodeBroker [53] and SnipMatch [50]. Strathcona defines three different types of structural queries : class, method or field. Their similarity is based out of six structural heuristics and does not incorporate language. CodeBroker's objective is to promote reuse of methods by avoiding redundant copies. Each time user writes javadoc for a new method, CodeBroker queries its database for methods with similar Java documentation. As soon as the signature of the method is available, ranking adapts to incorporate signature matches (Figure 3). It's limitation is that it heavily depends on good quality documentation and does not consider internal structure. SnipMatch is an eclipse plugin that introduces a markup language for code snippets. The markup language is used for indexing snippets to the repository that contains placeholders for variables. When a user queries for snippets, SnipMatch uses the markup to not only determine which snippet is relevant but also incorporates the snippet by applying the markup to the current context (Figure 4). This however creates a dependency on user participation in creating and indexing marked up snippets.

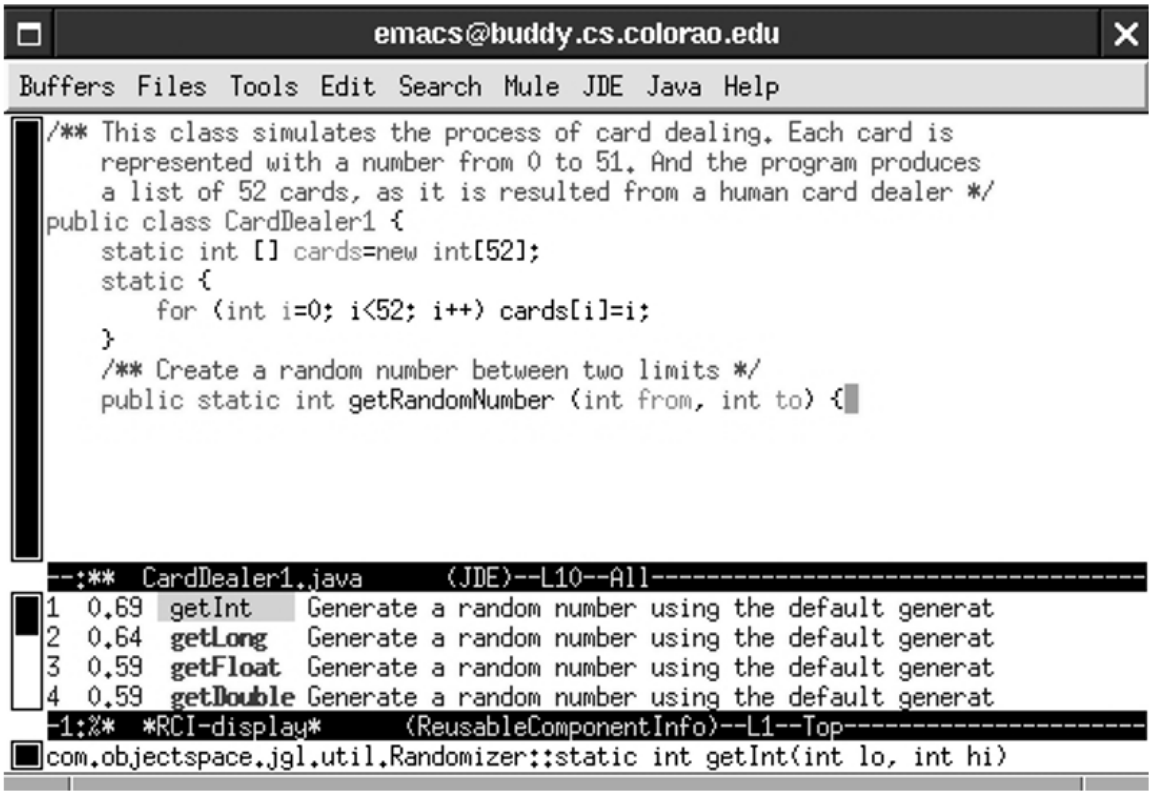


Figure 3: CodeBroker Recommendation Example



Figure 4: SnipMatch Recommendation Example

Chapter 3

CODERECO SYSTEM

3.1 Scope

In this thesis, Java methods are the *items* for recommendation. Java is chosen as the language of choice because as of today, it ranks first in popularity indexes [33, 46] and has the second most volume of tags in stackoverflow [34]. The choice of context size being *methods* is deliberate. Choosing smaller units of scope like partial snippets would have increased complexity of determining the boundaries of the amount of information that is relevant. Methods are considered as self-contained abstraction units in Java programming language. Developers write methods to encapsulate *tasks*, and execution of a program is the execution of a sequence of methods. Method similarity can be an important sub-component of *class* level similarity.

3.2 Design Rationale

The core component of CodeReco's design is it's code similarity metric. Based on background literature, common approaches for modeling code similarity are sequence of tokens, abstract syntax trees, concept or flow graphs and vector space models. For choosing an appropriate model for an *online* (synchronous) recommender system with a potentially large corpora, speed is critical and recommendations should be as *real-time* as possible. For methods A and B, finding similarity between sequence of tokens (string edit distance) has $\mathcal{O}(n_A \cdot n_B)$ time complexity where n is the number

of tokens in a method [48]. Detecting similarity between abstract syntax trees (tree edit distance) has $\mathcal{O}(n_A \cdot n_B \cdot min(depth(A), leaves(A)) \cdot min(depth(B), leaves(B)))$ where n is the number of nodes in an AST [55]. Graph based approached are NP hard [54] and exponential in some approximations. Vector space models (VSM) [40] have linear time complexity as distance metrics like euclidean or jaccard can be found in $\mathcal{O}(d)$ time where d is the number of dimensions of the vectors. Their simplicity and scalability makes VSM's popular models for application in large scale information retrieval and machine learning [27]. CodeReco uses the VSM model by converting each method into a vector in an engineered dimension space.

In natural language, Term-Frequency and Document Frequency weighted vector models are used to transform boolean frequency vectors to weighted term vectors [47]. The idea is that it is not only important to count the terms present in a document, it is also important to discount common terms across documents as they represent little information unique to a particular document. In terms of source code, an example would be that keywords like 'int' or 'main' present less unique information than words like 'StringBuffer' or 'Scanner'. In order to extract high level semantic information, VSM's can be used to apply advanced statistical techniques like LSA [8] and LDA [4] to deduce *topics* within documents. However, in the case of source code documents, language based semantics like LSA and LDA are not enough. Consider an example of two statements:

```
int id = 5;

String name = "John";
```

Term frequency models would consider them to be completely different and unrelated statements (no match). But at a structural-semantic level, both can be considered as variable decelerations and assignments. *Incorporating structural-semantic information*

on a scalable VSM is the underlying design rationale of CodeReco. The features of its VSM (and their high level categorization) are represented in table 1 with an example shown in figure 5. Features are clustered into categories for making personalization intuitive and less complicated. For example, the signature features of a vector would be the set of:

$$v_m(signature) = \{v_m[`params'], v_m[`returnType'], v_m[`modifier']\}$$

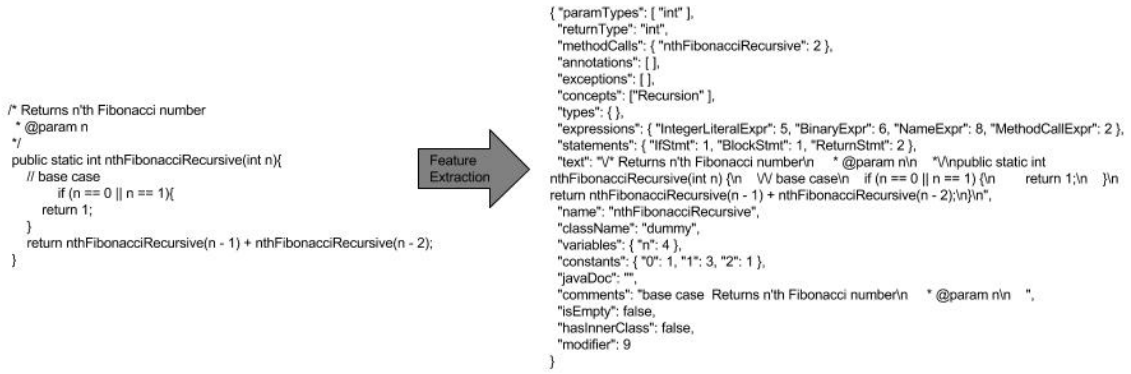
The open source javaparser[5] library is used for extracting structural-semantic features and separating language tokens with programming syntax tokens (as of February 2017, javaparser had a bug in differentiating java documentation with comments, but that does not affect CodeReco). This allows precise tokenization without having to build complex regular expression patterns specific to the programming language. The AST data structure created by javaparser is *flattened* (only counts of node types extracted) by CodeReco during traversal. Even though flattening results in loss of all sequential information, it still adds value in populating similarities in counts of structural components [15].

Table 1: CodeReco Vector feature categories

Signature	**Structure**	**Language**	**Concepts**
params	expressions	method_name	concepts
returnType	statements	variable_names	
modifier	methods_called	constants	
	exceptions	comments	
	annotations	java_doc	
	types	types	

[5]http://javaparser.org/

Figure 5: Feature Extraction



Types are considered in both 'Structure' and 'Language' because structurally they represent classes used; whereas with a camel case split, they represent the nomenclature of the class in language (e.g.: 'StringBuffer' acts as a buffer containing string). CodeReco is written in Python (Flask), with the feature extraction submodule based on the Java. It is a RESTFUL web service that returns JSON (JavaScript Object Notation) results, making it easy for anyone to query if they know the API. The high level architecture of CodeReco is given in figure 6

3.2.1 Similarity Metric

Jaccard similarity between two sets A and B is given as:

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|} \tag{3.1}$$

Cosine similarity between two vectors x and y of the same dimensions is given as:

$$C(x, y) = \frac{x \cdot y}{||x|| \cdot ||y||} \tag{3.2}$$

CodeReco's similarity metric S between any two method vectors is defined as:

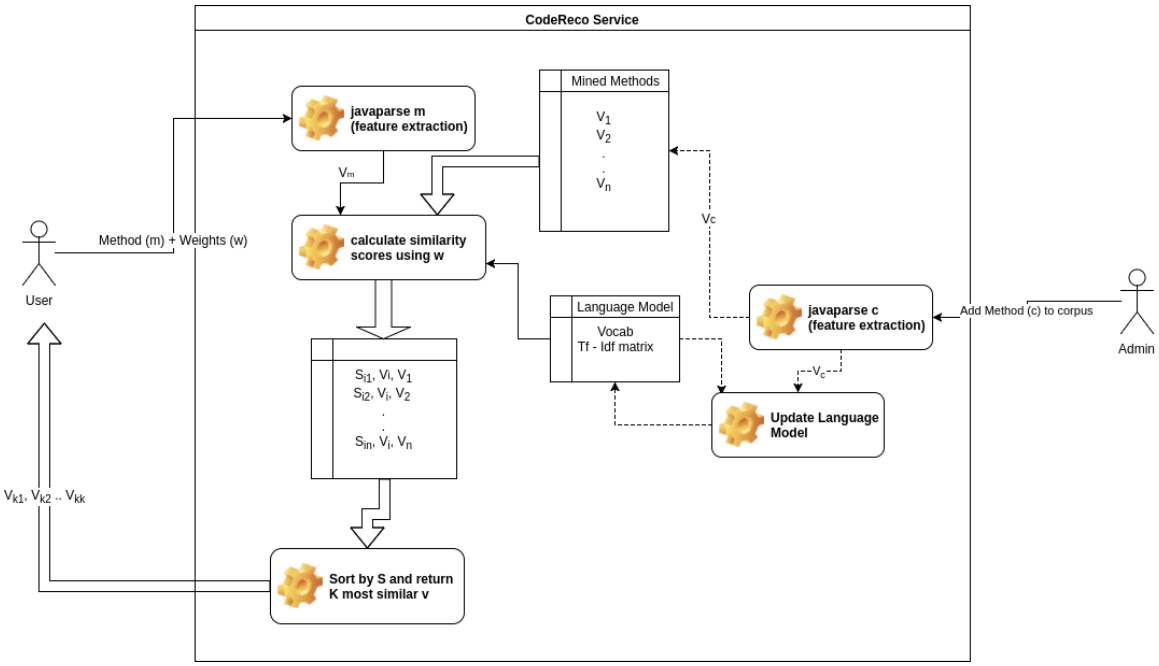
$$
\begin{aligned}
S^{w}_{CodeReco}(v_{m1}, v_{m2}) = & [w_{signature} \cdot J(v_{m1}(signature), v_{m2}(signature)) + \\
& w_{structure} \cdot J(v_{m1}(structure), v_{m2}(structure)) + \\
& w_{language} \cdot C_{tf-idf}(v_{m1}(language), v_{m2}(language)) + \\
& w_{concept} \cdot J(v_{m1}(concept), v_{m2}(concept))]/4
\end{aligned}
\tag{3.3}
$$

C_{tf-idf} represents cosine similarity (equation 3.2) where the value of the terms in the vectors are determined via a term frequency - inverse document frequency model [6]. This model is analogous to the default model used in open source search engines like lucene[7].

[6]https://radimrehurek.com/gensim/models/tfidfmodel.html

[7]https://www.elastic.co/guide/en/elasticsearch/guide/current/practical-scoring-function.html

Figure 6: CodeReco Design Architecture



3.2.2 Preprocessing

All empty methods are ignored. String representing numbers like 'zero' and 'one' are replaced by a sentinel value 'DIGIT'. Similarly, words like 'first' and 'second' are replaced by sentinel 'RANK', 'Hundred' or 'thousand' are replaced by 'SCALE' and 'January' or 'February' are replaced by 'TIME'. The NLTK [8] library is used for stop-word removal and lemmatization. Camel case words are split, i.e., 'rankedStudents' would be split as 'ranked' and 'students'.

3.2.3 Heuristics for Concept Extraction

All features except *concepts* are directly extracted from the javaparser AST. In order to capture more semantic information, concepts are extracted based on the following heuristics:

1. Recursion: a method call with the same name as the method is detected.
2. Polymorphism: a statement contains an instanceof or casting expression, or when a creation expression does not match the type of the deceleration expression.
3. Casting: there is a casting expression present
4. ExceptionHandling: try-catch statement is observed. Note that this is different than throwing exceptions
5. Synchronization: synchronized statement observed.
6. InnerClass: new class or interface deceleration is noticed.
7. InnerMethod: new method deceleration is noticed.

[8] http://www.nltk.org/

3.2.4 Algorithm and Complexity

CodeReco contains a mined corpus of method vectors added by an administrator. Given a method m, a weight vector w_i and configuration k, CodeReco first calculates similarity scores $S_{CodeReco}$ between m and each method vector in its corpus using w_i. It recommends the k most similar methods to the user. In other words, it returns the k *nearest* methods in the engineered dimension space based on w_i, analogous to item based collaborative filtering techniques used in recommender systems [42].

End to end algorithmic steps for recommending methods given a method with text t and weights w_i:

1. Extract vector v_m from queried text t using javaparser and heuristics
2. Create a list l of $[v_m, v_j, S^{w_i}_{CodeReco}(v_m, v_j)]$ for each v_j in corpus
3. Reverse sort l based on the S^{w_i}
4. Return $l[0...k][v_j]$

Time complexity of extracting v_m from t (step 1) is linear in size of t $\mathcal{O}(|t|)$ [11]. Step 2 is $\mathcal{O}(n \cdot d)$ where n is the number of vectors in corpus and d is the dimensionality of vectors. Sorting in step 3 is $\mathcal{O}(n \cdot \log_2 n)$ while step 4 is $\mathcal{O}(1)$. This makes the time complexity of CodeReco $\mathcal{O}(n \cdot (\log_2 n + d))$. For sufficiently large n and a fixed d, CodeReco has is $\mathcal{O}(n \cdot \log_2 n)$ time complexity, making it highly scalable.

3.2.5 Configuration

The default configuration of weights, unless changed by the user are:

$$w \leftarrow [w_{signature} = 1, w_{structure} = 1, w_{language} = 1, w_{concept} = 1]$$

Users can change individual feature-category weights on a scale of 0 to 5, and it will accordingly change similarity scores and return fresh results that may or may not change.

CodeReco will return 5 recommendations by default, unless changed by the admin.

$$k \leftarrow 5$$

3.3 Evaluation Methodology

Recommender systems are typically evaluated on desirable properties like usage prediction, rating or ranking predictions, diversity, coverage, scalability, utility, etc. These properties are measured via offline experiments, user studies or online experiments [44]. There are several challenges to verify the hypotheses stated in this thesis.

First of all, there are no existing ground truth datasets. To the best of my knowledge, there is no existing dataset that measures indirect *helpfulness* of method recommendations, thus ruling out most offline simulation techniques. Having no existing user base also rules out online experiments for measurement (A/B testing). User study is thus the chosen technique in this work, acknowledging their limitations [44].

Secondly, implicitly evaluating H2, H3 and H4 is a challenge. An ideal scenario would be to divide users randomly into groups that have access to CodeReco vs groups which don't, and evaluate metrics like time taken to complete the same programming tasks. Given the constraints of a having less than 20 users in a class, this could not reliably be done and had to be measured explicitly. Measuring *usage* explicitly is another challenge in our case. Unlike the case of e-commerce or movie recommendations, where it is straightforward to measure if a recommended item (movie) was used (watched)

by the user, defining usage from a Java method recommendation is non-trivial if the method is not directly invoked. Capturing copy actions from the recommendations is one possible approach to encapsulate the use of a recommended method. However, it may not be reliable enough as users can incorporate partial snippets, find ideas or hints helpful at a glance without explicit copying. Thus, to measure recommendation usage in CodeReco, users are asked to explicitly *rate* method recommendations based on how much they found it helpful in completing a programming problem, on a scale of 1 to 5. Additionally, they are not forced to rate all recommendations and non-rated methods are assumed to be irrelevant.

For testing H1 and H4, a *baseline* similarity metric was defined that treats Java methods as pure text. The same tf-idf based vector space model that is used in CodeReco is also used in Baseline (C_{tf-idf}). There are two key differences between them, the first being that Baseline considers the whole method as text, whereas CodeReco considers only those features listed in the 'language' category (table 1) as text. For example, keywords like *void*, *int* and method calls like *System.out.println* are considered as text in baseline but not in CodeReco. Second being that Baseline does not have any structural or semantic information. Note that advanced techniques like LSI and usage of lexical databases like WordNet [30] are skipped for simplification because the goal of H1 and H4 is to evaluate benefits of a combined model (structure-semantics with language) as compared to a Baseline (language).

H1 is evaluated experimentally with the help of a concept indexing service JavaParser [17] (different from javaparser used for feature extraction). JavaParser extracts concepts from Java code based on a defined ontology of concepts [9]. It reports 93% overlap with manual indexing of Java code for that ontology, and can thus be con-

[9]http://www.pitt.edu/~paws//ont/java.owl

sidered as an automated equivalent of a manual expert indexing method samples for concepts from that ontology. 1852 Java method examples from the first and second versions of the book 'Java Examples in a Nutshell'[10] were used as a test dataset.

H4 is evaluated by using both *sources* of recommendations : CodeReco and Baseline in the user study. 5 nearest methods from CodeReco and Baseline are recommended to the users, randomly shuffled on each query. An interesting configuration is that if a user sets CodeReco weight vector as all 0 except language, would CodeReco be equivalent to baseline? It is not so because even though both would be purely tf-idf models, their corpus of text is different.

H2 and H3 are subjective questions asked to users post participation in a survey (feedback).

[10]http://examples.oreilly.com/9781565923713

Chapter 4

USER STUDY

4.1 Recommendation Corpus

6214 compilable methods are crawled from Java textbook code samples and other online learning resources like Big Java 6th Edition [11], Java CookBook 3rd edition [12], UT Austin CS307 [13], Head First Java [14], Oracle Press Java Programming [15], HWS java notes [16].

4.2 Study Design

A web based application was implemented for the methodology described in section 3.3. The study flow is presented in figure 7.

Users were asked to write methods for solving below two problems:

[11] http://horstmann.com/bigjava.html

[12] https://github.com/oreillymedia/java_cookbook_3e

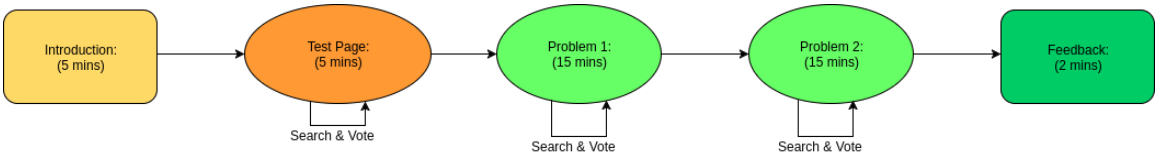
[13] https://www.cs.utexas.edu/~scottm/cs307/codingSamples.htm

[14] http://www.headfirstlabs.com/books/hfjava/

[15] https://www.mhprofessionalresources.com/getpage.php?c=oraclepress_downloads.php

[16] http://math.hws.edu/javanotes/

Figure 7: User Study Workflow



1. Write a method that takes an integer array and returns the largest difference between its elements.
   For example, if array contains [7, 4, 1], the method should return 6 as the largest difference (7 − 1)

2. ```
   class Person{
       String name;
       int age;
   }
   class Employee extends Person{
       double getSalary();
   }
   class BusinessOwner extends Person {
       double getProfit();
   }
   class Veteran extends Person{
       String veteran_id;
   }
   ```

   Write a java method that takes a person and returns their tax as below:
   1. For employees, if salary < $10,000, 2% of salary, else 5%
   2. For business owners, 10% of their profit
   3. 0 for veterans

   What would you do if a person is none of these?

Both problems were designed with caution for no trivial matches (method doing the exact task) present in the corpus. Problem 1 was designed as an enhancement of a common sub-problem (finding maximum or minimum element) to test applied
   ```

algorithmic skills. Problem 2 was designed as an application of object oriented concepts within a method.

Half of the users were assigned problem 1 and half problem 2 to start the study. While a user is assigned one problem, they could not attempt the other until they submit their solution. This was done in order to avoid familiarity bias towards any particular problem. While participants were writing their methods, they could search for recommendations as shown in figure 8. The only prerequisite for getting recommendations was that their text had to be parsable (no syntax errors).

A maximum of 10 recommendations are shown for each search query, consisting of 5 nearest samples from both CodeReco and Baseline. These are randomly shuffled with no visual cue to differentiate between the source model. Users were asked to rate these recommendations on a scale of 1 to 5, based on how much they thought the recommendation helped them complete their current task (figure 9). In cases of overlap between recommendations from both algorithms, the method was shown only once but records were maintained for both source models separately, as their ranks (k value in k nearest) might be different between the algorithms.

User studies are expensive to conduct and tend to have fewer participants than ideal. In order to collect enough votes for analysis, at least 15 ratings (votes) were asked for each problem before they could submit and proceed to the next problem. The study was designed to be completed by students in a class within 45 minutes, and subjective feedback was collected towards the end. Users were informed not to use search engines to avoid ratings to be influenced by external search results. For evaluating H3, users should spend some time customizing their query in CodeReco instead of conventionally querying the web.

Figure 8: Searching

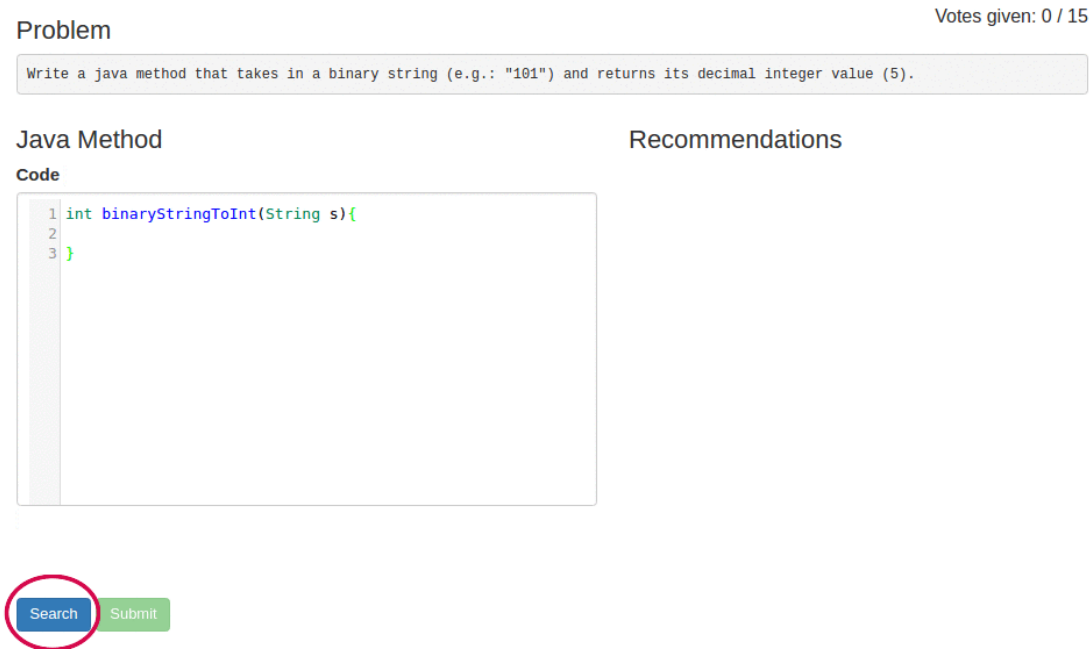


Figure 9: Rating (Voting)



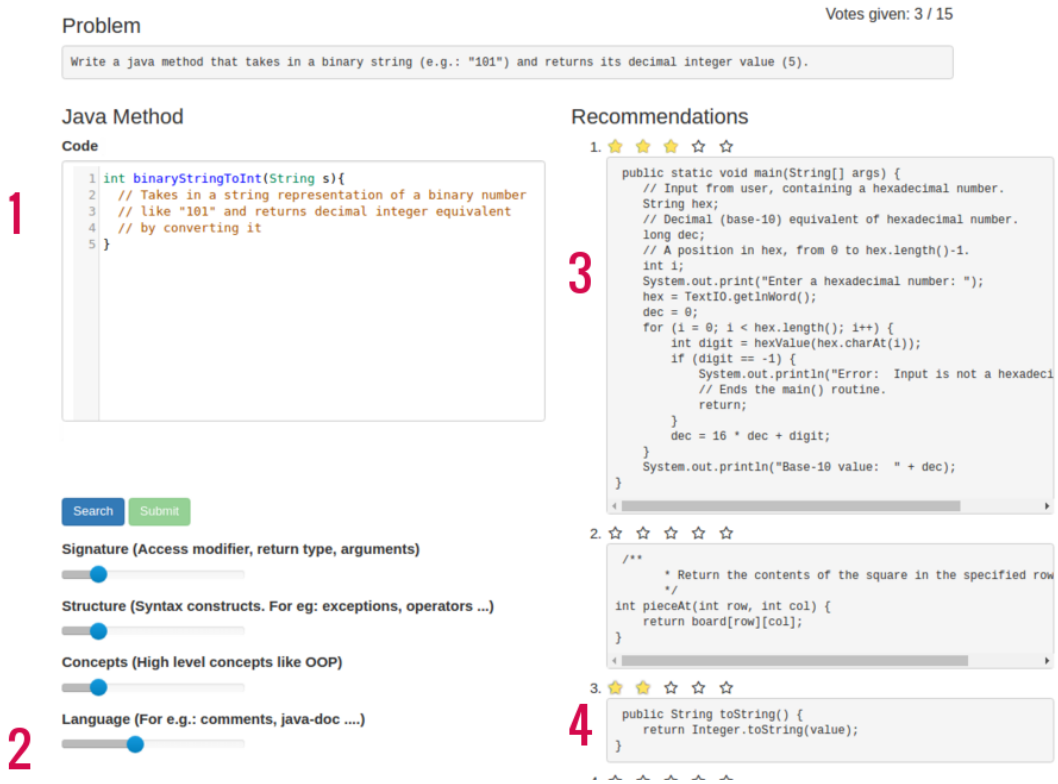
Users were also allowed to customize the weights of the queries as shown in the figure 10. For any change in the weights, a fresh query was executed with updated weights. An example of customization is shown in figure 10, where the user first adds comments in his Java method trying to summarize what they need to do (1). By

increasing weights in the *Language* slider, they are essentially adding more weight to language in order to boost matching keywords in comments or variable names. They find (3) and (4) as relevant methods, (3) being related to conversion from hexadecimal to base 10. It is not exactly what they are looking for, but can act as a reference algorithm. On the other hand, (4) seems to be an easier and straight forward way to convert, but it returns a String. Maybe by exploring the Integer class a little more, the user may find an even more relevant method with the right signature by boosting the signature component.

CodeReco was first used by novices at undergraduate freshmen classrooms of CPI101 (Introduction to Informatics) and ASU101 - CSE (ASU Experience in computer science and engineering) at Arizona State University. It was observed that a significant number of students were not comfortable with Java syntax and needed assistance in debugging. To get data from users comfortable with java syntax, the same study was also conducted on mechanical turk [17] (MTurk), a popular marketplace for human intelligent tasks. A qualification test was created (Appendix B) so as to select ideal workers for participation (beginners to medium competence). The test was designed so that qualifying participants will be able to debug simple Java methods without the help of compiler output and also have a better understanding of the features available for customization. 27 MTurk users qualified to participate, their average Java programming experience being close to 4 years.

[17] https://www.mturk.com/

Figure 10: Customization



4.3 Usage Descriptive Statistics

Descriptive statistics in table 2 provide some insights on participant behavior. MTurk users did not have any instance of copying text from the problem statement itself and much higher word counts in their submissions. They also explored CodeReco customizations more extensively than their classroom counterparts (Figure 11). Conceptual features were least explored in the classroom, which seems reasonable considering their lack of experience. We also visualize the distribution of the sub-set of CodeReco weights that corresponded to votes in figure 12. User searches as compared to their current word count is plotted in figure 13.

Table 2: UserStudy Descriptive Statistics

	Classroom	**MTurk**
Users	37	27
Total Ratings	807	773
Ratings – Baseline	2.33 ± 1.36	2.08 ± 1.24
Ratings – Codereco	1.88 ± 1.23	2.04 ± 1.23
Total Customizations	1045	1355
Signature	1.62 ± 1.13	2.07 ± 1.53
Structure	1.5 ± 1.03	2.15 ± 1.53
Concepts	1.27 ± 0.95	2.0 ± 1.56
Language	1.77 ± 1.36	2.14 ± 1.58
Copies from Question	3	0
Copies from Baseline Reco	13	11
Copies from Codereco Reco	6	6
Avg words per submission	29	49

Figure 11: CodeReco - Exploratory customizations

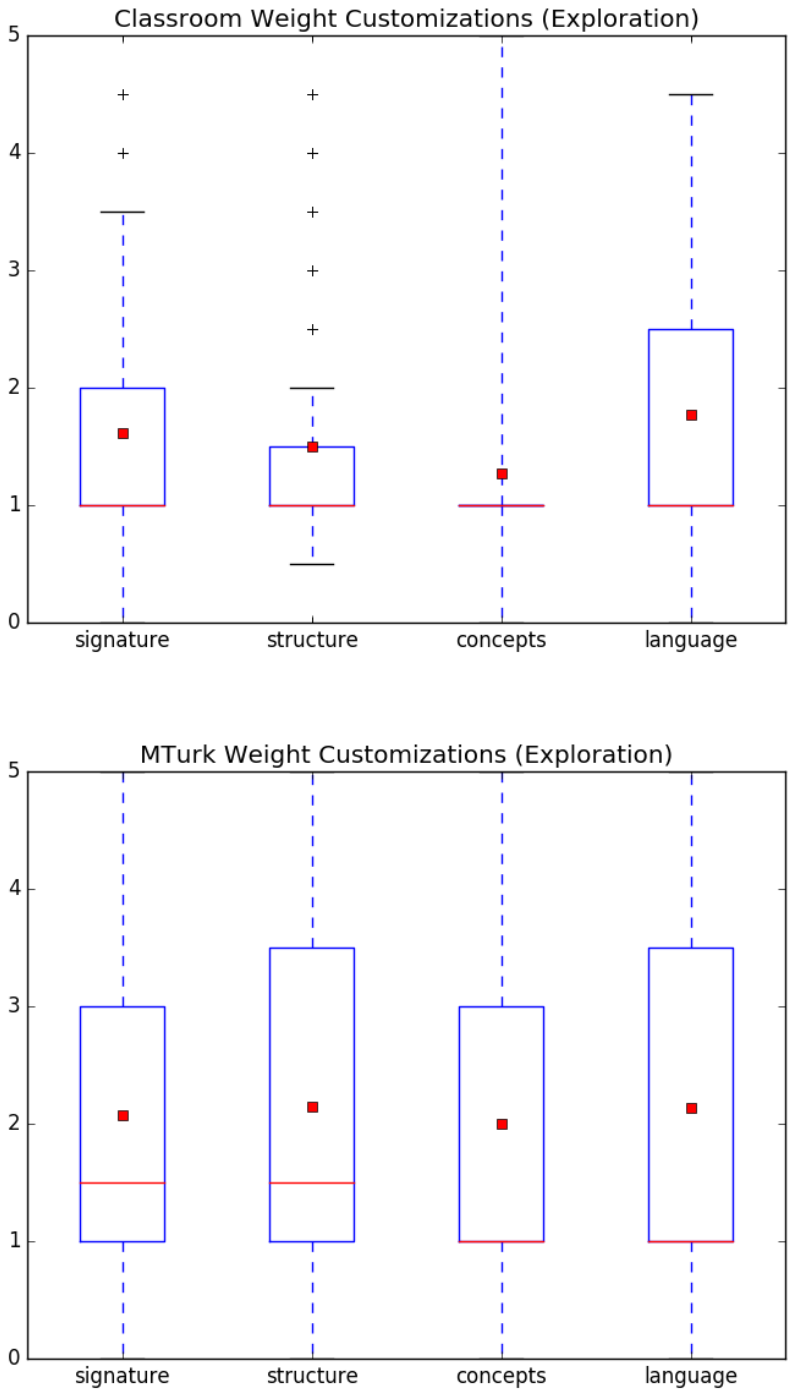
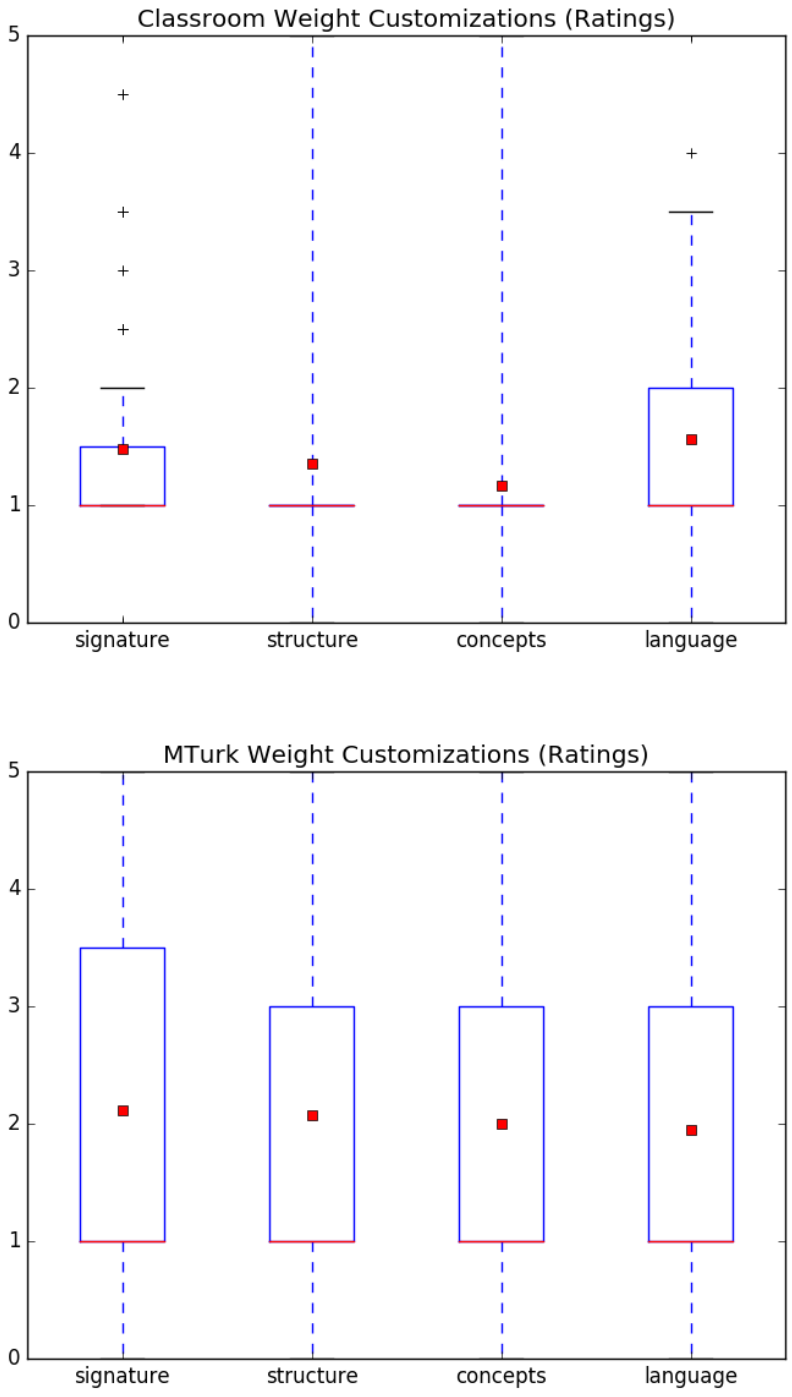


Figure 12: CodeReco - Rating Customizations



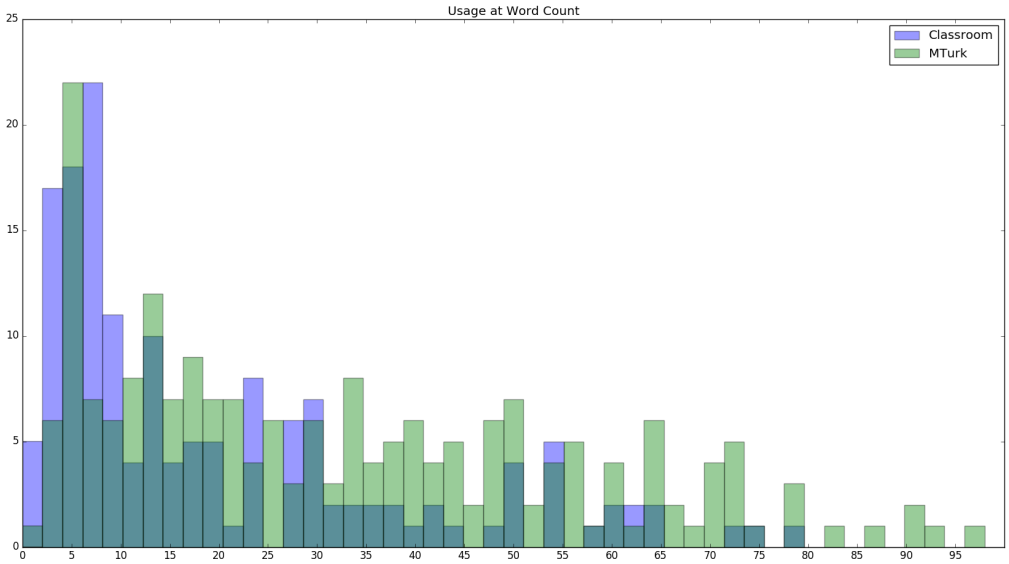


Figure 13: CodeReco Number of Queries at Word Counts

Chapter 5

EVALUATION

5.1 Evaluation Metrics

Challenges in using precision and recall or RMSE based evaluation metrics have been discussed in section 3.3. The rationale for some metrics used are presented below:

Shannon's Entropy H is a common measure of diversity (distributional inequality) in datasets [44].

$$H = -\sum_{i=1}^{n} p(i) \log p(i)$$

This is used in for for measuring diversity of ontology concepts covered in recommendations. Higher H values imply a distribution is more informative (has diverse components).

If J (Equation 3.1) represents jaccard similarity between ontology concepts of the test samples with respect to a recommendation, higher J implies more conceptual similarity [41].

5.2 Evaluation Results

CodeReco yields broader programming concepts

$H_{Baseline} = 1.51 \pm 0.8$ and $H_{CodeReco} = 1.70 \pm 0.82$. Students paired T test showed the distributions to be significantly different with $p < 0.001$.

CodeReco yields higher conceptual similarity with samples

$J_{Baseline} = 0.20 \pm 0.23$ and $J_{CodeReco} = 0.35 \pm 0.30$. Students paired T test showed the distributions to be significantly different with $p < 0.001$.

Baseline yields better ratings in classroom

Classroom ratings for Baseline were significantly higher than CodeReco ($p < 0.001$ as per Mann-Whitney U test [10]) [18]. There were insignificant differences in ratings for MTurk users ($p > 0.5$).

Feature correlations

Linear correlations between feature categories vs ratings for CodeReco are summarized in tables 3 and 4.

In the classroom, signature - language and structure-concepts had significant linear correlations ($p < 0.001$), that might indicate that students looked for a combination of (signature - language) or (structure - concepts). The rating-weights (Figure 12) also suggest that signature and language were most sought by classroom students. There were insignificant correlations between features and ratings.

MTurk users had significant linear correlations between signature - structure - concepts ($p < 0.001$), showing that they indeed consider all three as belonging to a higher category of *structure-semantic* features. The strongest amongst all correlations

[18] Mann-Whitney U test is used for ratings instead of the more widely used Student t-tests to avoid normal distribution assumption of ratings

were between (structure - concepts). There were also significant correlations between structure with ratings and concepts with ratings ($p < 0.001$), indicating that an increase in these features likely increased ratings of CodeReco recommendations.

Table 3: Classroom CodeReco Weight Correlations

Pearson Corr (r)	**signature**	**structure**	**concepts**	**language**	**rating**
signature	1	0.14	0.10	0.30	0.08
structure	0.14	1	0.31	0.14	-0.02
concepts	0.10	0.31	1	0.11	-0.03
language	0.30	0.14	0.11	1	0.00
rating	0.08	-0.02	-0.03	0.00	1

Table 4: MTurk CodeReco Weight Correlations

Pearson Corr (r)	**signature**	**structure**	**concepts**	**language**	**rating**
signature	1	0.44	0.39	0.29	0.09
structure	0.44	1	0.53	0.17	0.21
concepts	0.39	0.53	1	0.23	0.27
language	0.29	0.17	0.23	1	0.04
rating	0.09	0.21	0.27	0.04	1

Subjective feedback

At the end of both classroom and MTurk activities, users were asked to fill a short subjective survey on their experience (visualized in Figure 15) and optional comments (listed in Appendix 5). Classroom participants fell less sure about recommendations reducing search time, whereas MTurk users felt less sure about customizations being intuitive. Many users complained about the lack of syntax debugging support. 61.5% users had a positive (non-neutral) response on recommendations being helpful. 50% users had a positive response on recommendations reducing search time.

Some thoughtful excerpts from textual feedback were:

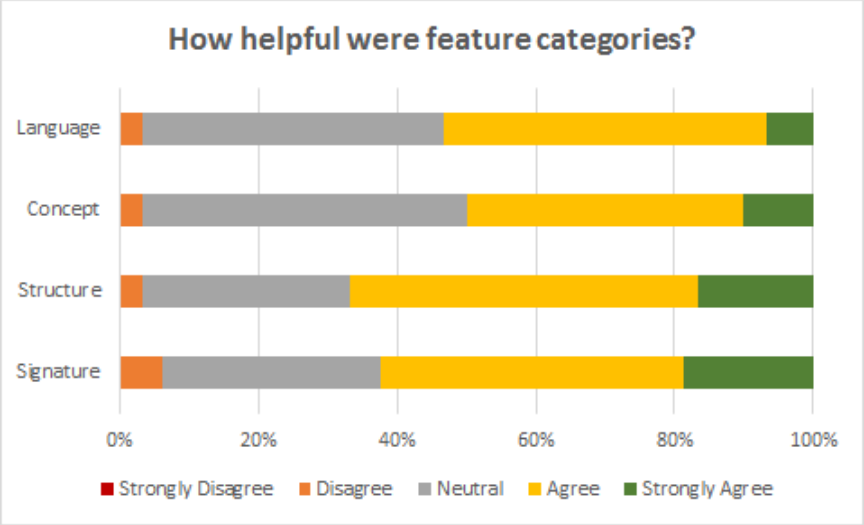
'Overall the idea is fabulous, however textual hints or points might help better for beginning programmers compared to actual code that your system was recommending. This is because, for a beginning programmer, reading recommended code in itself becomes an exercise. You can try putting shorter more relevant hint kind of features compared to such larger chunks of code being recommended'

'It legitimately lowered the time it would have taken to complete both questions by a large amount. I'm impressed with how similar the searches were without even having to change the sliders much. But the capability to do so gave the ability to fine tune it. I wish I had this for my past Java assignments!'

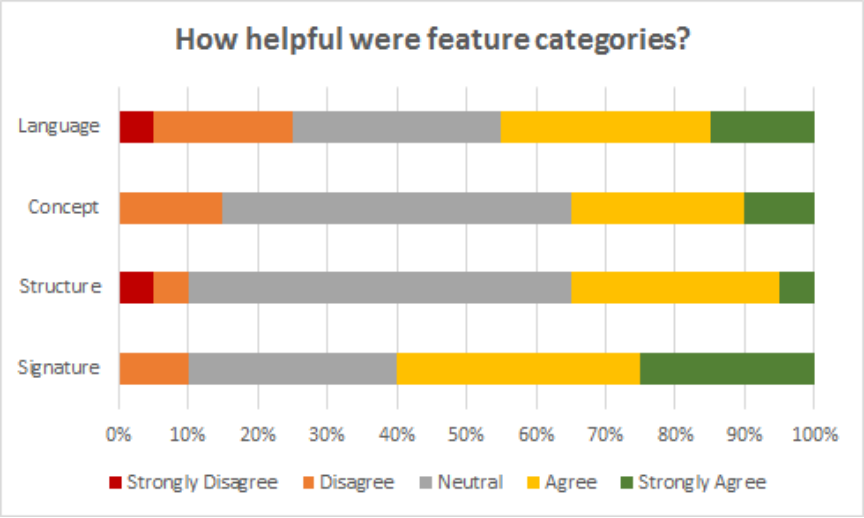
'Seems to work OK when searching for general algorithms, but falls a bit short on specific structures'

'I like the concept for sure, if it had a little more problem specific recommendations I'd be much more keen to use it'

Figure 14: Features feedback

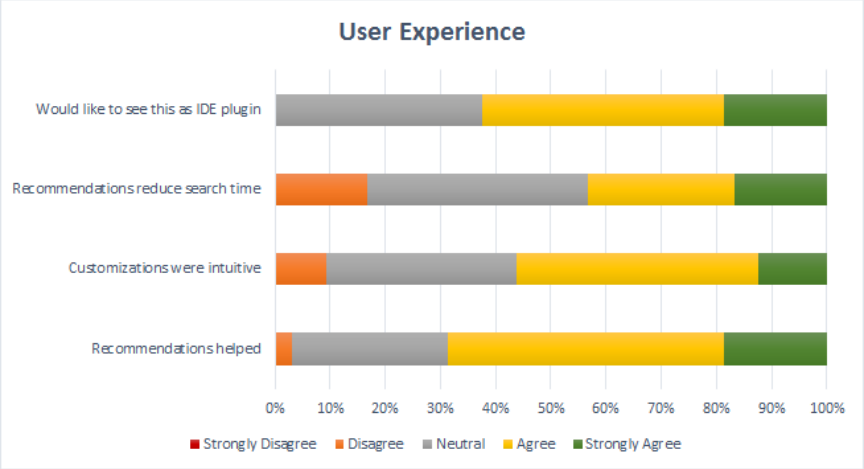


Classroom

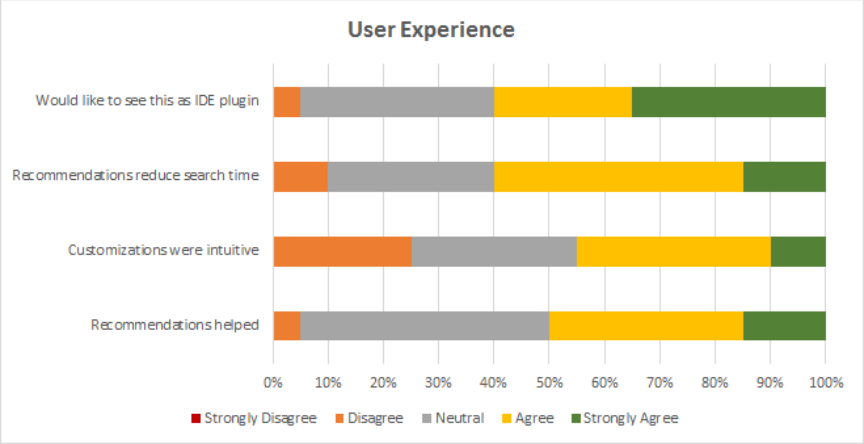


MTurk

Figure 15: Subjective User experience



Classroom



MTurk

Chapter 6

CONCLUSION

The purpose of this thesis was two fold, to evaluate a novel approach of sampling semantically similar code samples and to estimate the empirical utility of such recommendations (H1-4 in section 1.2). Java methods were chosen as the scope and CodeReco was designed to provide semantic Java method recommendations in a scalable vector space model. It segregates language and syntax from methods based on its parsed AST and extracts structure-semantic features. An original similarity metric was designed to provide intuitive categorization of features and enable user personalization. User study was conducted on classrooms and mechanical turk for evaluating the hypotheses. User studies have time constraints, user biases, and the two problems asked cannot cover the vast space of relevant information seeking scenarios. However, given its constraints, it is still an effective measure of empirical value [44].

6.1 Discussions and Summary

As per evaluation results in section 5.2, an ontology based indexing service verifies H1; i.e, as compared to a language baseline, CodeReco recommendations had a broader coverage of concepts and were also conceptually more similar. H2 and H3 were subjective hypotheses that received positive feedback. 61.5% users believed that method recommendations are helpful and 50% users believed they would potentially reduce their search times. This shows that users are open to finding utility in recommendations that are not syntactically applicable but semantically similar.

However, a counter-intuitive finding was that H4 (CodeReco being more helpful than Baseline) is rejected. A possible explanation could be that participants who lack experience would depend more on language for information seeking, whereas participants who understand structural semantics can leverage both for seeking information (figure 11). Figure 13 shows that users sought most recommendations initially with fewer words written (and thus, fewer structural components). An example scenario where language model outperforms CodeReco was when users did not know how to use 'instanceof' operator and were facing syntax errors. CodeReco would only detect 'instanceof' if it was correctly used in an example as an operator. However, the language model recognizes 'instanceof' as just another *term*, and any occurrence in comments is equivalent to a syntactically correct usage, thus producing an important match. Another possible reason could be that novice users may not be prompt in correlating semantic similarities between recommendations and their method, deeming more *literal* matches as helpful. In the case of experienced users, there were significant correlations between structure-semantics with ratings, indicating that they could identify and appreciate structure-semantic similarities.

As a system, CodeReco is scalable and modular with a JSON interface. Since it depends only on parsed AST's and not compiled code, it is possible to extend the use of CodeReco across other Java-like languages like C++.

6.2 Limitations and Future Work

CodeReco's major limitation is its dependency on correct syntax for feature extraction, which is hard for novices to achieve without support. This can be overcome with IDE integration. Another limitation observed was lack of inverse document

frequency weights for the structural-semantic features. Currently, CodeReco gives an 'ObjectMethodInvocation' and a 'ThisReference' equal weights (only counts), whereas in the corpus the latter may be much rarer than former. Instead of partial segregation of language and structure tokens, all tokens can be considered in both feature-sets to cover for cases just discussed.

Controllable personalization is empowering for users but it also brings the same information seeking challenges in a different form, they might not know what is an optimal configuration for what they seek. There are two possible ways to mitigate this. CodeReco could be optimized for specific objectives instead of user utility, like acting as an intelligent tutoring system. Weights can be trained by optimizing against a Java Tutor's method recommendations. Alternatively, there is scope for incorporating user modeling where the system optimizes its weights for the user's utility based on implicit feedback or explicit ratings.

REFERENCES

[1] James Allan et al. “Challenges in information retrieval and language modeling: report of a workshop held at the center for intelligent information retrieval, University of Massachusetts Amherst, September 2002”. In: *ACM SIGIR Forum*. Vol. 37. 1. ACM. 2003, pp. 31–47.

[2] Ira D Baxter et al. “Clone detection using abstract syntax trees”. In: *Software Maintenance, 1998. Proceedings., International Conference on*. IEEE. 1998, pp. 368–377.

[3] Nicolas J Belkin. “Helping people find what they don’t know”. In: *Communications of the ACM* 43.8 (2000), pp. 58–61.

[4] David M. Blei, Andrew Y. Ng, and Michael I. Jordan. “Latent Dirichlet Allocation”. In: *Journal of Machine Learning Research* 3 (2001), pp. 993–1022.

[5] Kevin W Bowyer and Lawrence O Hall. “Experience using" MOSS" to detect cheating on programming assignments”. In: *Frontiers in Education Conference, 1999. FIE’99. 29th Annual*. Vol. 3. IEEE. 1999, 13B3–18.

[6] Sergey Brin, James Davis, and Hector Garcia-Molina. “Copy detection mechanisms for digital documents”. In: *ACM SIGMOD Record*. Vol. 24. 2. ACM. 1995, pp. 398–409.

[7] Scott C. Deerwester et al. “Indexing by Latent Semantic Analysis”. In: *JASIS* 41 (1990), pp. 391–407.

[8] Scott Deerwester et al. “Indexing by latent semantic analysis”. In: *Journal of the American society for information science* 41.6 (1990), p. 391.

[9] *Eclipse Intelligent Code Completion*. http://www.eclipse.org/recommenders/manual. Accessed: 2017-05-05.

[10] Michael P Fay and Michael A Proschan. “Wilcoxon-Mann-Whitney or t-test? On assumptions for hypothesis tests and multiple interpretations of decision rules.” In: *Statistics surveys* 4 (2010), pp. 1–39.

[11] Bryan Ford. “Packrat parsing:: simple, powerful, lazy, linear time, functional pearl”. In: *ACM SIGPLAN Notices*. Vol. 37. 9. ACM. 2002, pp. 36–47.

[12] William B Frakes and Brian A Nejmeh. “Software reuse through information retrieval”. In: *ACM SIGIR Forum*. Vol. 21. 1-2. ACM. 1986, pp. 30–36.

[13] Rosalva E. Gallardo-Valencia and Susan Elliott Sim. “Internet-Scale Code Search”. In: 2009.

[14] *Github Repositories*. https://api.github.com/repositories?since=80000000. Accessed : 2017-05-05.

[15] Maurice Howard Halstead. *Elements of software science*. Vol. 7. Elsevier New York, 1977.

[16] Reid Holmes and Gail C Murphy. “Using structural context to recommend source code examples”. In: *Software Engineering, 2005. ICSE 2005. Proceedings. 27th International Conference on*. IEEE. 2005, pp. 117–125.

[17] Roya Hosseini and Peter Brusilovsky. “JavaParser: A Fine-Grained Concept Indexing Tool for Java Problems”. In: 2013.

[18] Michelle Ichinco, Wint Hnin, and Caitlin Kelleher. “Suggesting examples to novice programmers in an open-ended context with the example guru”. In: *2016 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)* (2016), pp. 230–231.

[19] *Java WikiBook*. . Accessed : 2017-05-05.

[20] Andrew Jensen Ko, Brad A. Myers, and Htet Htet Aung. “Six Learning Barriers in End-User Programming Systems”. In: *2004 IEEE Symposium on Visual Languages - Human Centric Computing* (2004), pp. 199–206.

[21] Carol Collier Kuhlthau. “Inside the search process: Information seeking from the user’s perspective”. In: *JASIS* 42 (1991), pp. 361–371.

[22] Adrian Kuhn, Stéphane Ducasse, and Tudor Gîrba. “Semantic clustering: Identifying topics in source code”. In: *Information and Software Technology* 49.3 (2007), pp. 230–243.

[23] Seonah Lee and Sungwon Kang. “A Study on Guiding Programmers’ Code Navigation with a Graphical Code Recommender”. In: *SERA*. 2011.

[24] Chao Liu et al. “GPLAG: detection of software plagiarism by program dependence graph analysis”. In: *KDD*. 2006.

[25] Dastyni Loksa and Andrew Jensen Ko. “The Role of Self-Regulation in Programming Problem Solving Process and Success”. In: *ICER*. 2016.

[26] Yuri Malheiros et al. “A source code recommender system to support newcomers”. In: *Computer Software and Applications Conference (COMPSAC), 2012 IEEE 36th Annual*. IEEE. 2012, pp. 19–24.

[27] Christopher D Manning, Prabhakar Raghavan, Hinrich Schütze, et al. *Introduction to information retrieval*. Vol. 1. 1. Cambridge university press Cambridge, 2008.

[28] Nikos Manouselis et al. “Recommender Systems in Technology Enhanced Learning”. In: *Recommender Systems Handbook*. 2011.

[29] Kim Mens and Angela Lozano. “Source Code-Based Recommendation Systems”. In: *Recommendation Systems in Software Engineering*. Ed. by Martin P. Robillard et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 93–130.

[30] George A Miller. “WordNet: a lexical database for English”. In: *Communications of the ACM* 38.11 (1995), pp. 39–41.

[31] Denis Parra and Peter Brusilovsky. “User-controllable personalization: A case study with SetFusion”. In: *Int. J. Hum.-Comput. Stud.* 78 (2015), pp. 43–67.

[32] Peter L Pirolli and John R Anderson. “The role of learning from examples in the acquisition of recursive programming skills.” In: *Canadian Journal of Psychology/Revue canadienne de psychologie* 39.2 (1985), p. 240.

[33] *PYPL Index*. http://pypl.github.io/PYPL.html. Accessed: 2017-05-05.

[34] *Question Tags*. http://stackoverflow.com/tags. Accessed: 2017-05-05.

[35] Chaiyong Ragkhitwetsagul, Jens Krinke, and David Clark. “A Comparison of Code Similarity Analysers”. In: *RN* 17.04 (2017), p. 04.

[36] Alexander Renkl. “Toward an Instructionally Oriented Theory of Example-Based Learning”. In: *Cognitive Science* 38 (2014), pp. 1–37.

[37] Francesco Ricci, Lior Rokach, and Bracha Shapira. “Recommender Systems Handbook”. In: 2015.

[38] Kelly Rivers and Kenneth R Koedinger. “Data-driven hint generation in vast solution spaces: a self-improving python programming tutor”. In: *International Journal of Artificial Intelligence in Education* (2015), pp. 1–28.

[39] Gerard Salton and Chris Buckley. “Term-Weighting Approaches in Automatic Text Retrieval”. In: *Inf. Process. Manage.* 24 (1988), pp. 513–523.

[40] Gerard Salton, A. Wong, and C. S. Yang. “A Vector Space Model for Automatic Indexing”. In: *Commun. ACM* 18 (1975), pp. 613–620.

[41] David Sánchez et al. “Ontology-based semantic similarity: A new feature-based approach”. In: *Expert Systems with Applications* 39.9 (2012), pp. 7718–7728.

[42] Badrul Sarwar et al. “Item-based collaborative filtering recommendation algorithms”. In: *Proceedings of the 10th international conference on World Wide Web*. ACM. 2001, pp. 285–295.

[43] Saul Schleimer, Daniel S Wilkerson, and Alex Aiken. “Winnowing: local algorithms for document fingerprinting”. In: *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*. ACM. 2003, pp. 76–85.

[44] Guy Shani and Asela Gunawardana. “Evaluating recommendation systems”. In: *Recommender systems handbook*. Springer, 2011, pp. 257–297.

[45] Giriprasad Sridhara et al. “Towards automatically generating summary comments for java methods”. In: *Proceedings of the IEEE/ACM international conference on Automated software engineering*. ACM. 2010, pp. 43–52.

[46] *TIOBE Index*. https://www.tiobe.com/tiobe-index/. Accessed: 2017-05-05.

[47] Peter D Turney and Patrick Pantel. “From frequency to meaning: Vector space models of semantics”. In: *Journal of artificial intelligence research* 37 (2010), pp. 141–188.

[48] Esko Ukkonen. “Algorithms for Approximate String Matching”. In: *Information and Control* 64 (1985), pp. 100–118.

[49] Jeremy Warner and Philip J. Guo. “CodePilot: Scaffolding End-to-End Collaborative Software Development for Novice Programmers”. In: *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*. CHI ’17. Denver, Colorado, USA: ACM, 2017, pp. 1136–1141.

[50] Doug Wightman et al. “Snipmatch: using source code context to enhance snippet retrieval and parameterization”. In: *Proceedings of the 25th annual ACM symposium on User interface software and technology*. ACM. 2012, pp. 219–228.

[51] Edmund Wong, Taiyue Liu, and Lin Tan. “CloCom: Mining existing source code for automatic comment generation”. In: *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE. 2015, pp. 380–389.

[52] Yunwen Ye. “An Empirical User Study of an Active Reuse Repository System”. In: *ICSR*. 2002.

[53] Yunwen Ye and Gerhard Fischer. “Reuse-Conducive Development Environments”. In: *Autom. Softw. Eng.* 12 (2005), pp. 199–235.

[54] Zhiping Zeng et al. “Comparing Stars: On Approximating Graph Edit Distance”. In: *PVLDB* 2 (2009), pp. 25–36.

[55] Kaizhong Zhang and Dennis Shasha. “Simple fast algorithms for the editing distance between trees and related problems”. In: *SIAM journal on computing* 18.6 (1989), pp. 1245–1262.

[56] Hao Zhong et al. “MAPO: Mining and Recommending API Usage Patterns”. In: *ECOOP*. 2009.

APPENDIX A

USER COMMENTS

Table 5: Classroom comments

Comments
avoid repeat recommendations when changing sliders
it would be helpful if we could search without correct code
Overall a good feature to have
Give more instructions on how to use the customizations. At first use, I wasn't exactly sure what they did.
Maybe we need more time to familiar that exercise.
in general, program is effective at helping, but some of the recommendations are not useful
I think the website requires more clarification on the instructions on how the features relate to the recommendations and also the instructions in general on how to submit.
If we had more time to participate, I could have gotten a better feel to it. Reduce the amount of recommendations shown at a time.
Rating the recommendations was a bit confusing and felt unnecessary
It will be great if there was a way to refresh the recommendations given as well as getting rid of some.
I personally couldn't really get the recommendations to appear unless I used the examples provided so take my feedback with a grain of salt. Good concept though.
I think the fact it doesn't tell you why your getting a compilation error is really hard for new people to programming. On other editors it tells you what line has the issue so you can figure out what to fix.
The interface was not super intuitive, could be easier to navigate.
Keep up the good work
Depending on what structure the user has, there could be different examples on different languages other than Java.
Recommendations were very helpful, but would like to see some visual improvements.
There were some problems submitting when you reached 15 stars in the study. Overall idea is great.
Seems like a good replacement for google just needs more fine tuning.
Suggestions without the code having to compile
show similar example
It legitimately lowered the time it would have taken to complete both questions by a large amount. I'm impressed with how similar the searches were without even having to change the sliders much. But the capability to do so gave the ability to fine tune it. I wish I had this for my past Java assignments!
The error message is always the same, but when someone inputs an error into the search and fix it they will likely see the same error message not knowing if their search is still faulty or if it is just the page loading.
The customization did not feel extremely intuitive but I would like to see this implemented and liked it overall.

Table 6: MTurk comments

Comments
avoid repeat recommendations when changing sliders
Could not get the recommendations to pop up anything with 'instanceof' or 'throw new Exception' , no matter how hard I tried to guide it. Seems to work OK when searching for general algorithms, but falls a bit short on specific structures. Interesting concept nonetheless. And btw, the link to this form in the test lead to a Restricted one.
I was not able to submit my recommendations, why ?
"Regarding the user study: 1. Navigation is a bit fuzzy. By reading the instructions and clicking start ""HERE"" link, I was directed to FAQs instead of Problem 1. On clicking Problem 1, a message comes to solve other problem first. 2. I assume that the recommendations are used to solve the problem. There were times when I needed an answer regarding syntax but I coudn't get a recommendation since syntax has to be right. 3. During problem solving, usually there are questions like ""how to sort an array using a library"", there seems to be no way to get recommendations for these questions. Writing in comments about this is similar to googling. 4. Points 2 and 3 are listed only because the user is forced to give 15 recommendations. With the given problems and I, after getting a few recommendations, I was forced to find the other recommendations and sometimes mark even if they were irrelevant. IMPORTANT: Some recommendations (3-4) were very pretty accurate 5/5 but rest were not related. Those 3-4 recommendations were enough to solve problem but I had to forcefully select other irrelevant recommendations. "
Had to submit before completion because I wanted to check the code for completion. Might be better to add some kind of verification that the questions were submitted.
Because sliders are used for the search features for Signature, Structure, etc. . . , it's a bit ambiguous as to how "strict" they make the search.
I like the concept for sure, if it had a little more problem specific recommendations I'd be much more keen to use it
"Honestly, the questions were too easy since I am a java instructor and we use these in our very first course. Secondly, some of your recommendations were completely irrelevant (i.e. what is the point of recommending a method with paintComponent in a question on array differences?). Overall the idea is fabulous, however textual hints or points might help better for beginning programmers compared to actual code that your system was recommending. This is because, for a beginning programmer, reading recommended code in itself becomes an exercise. You can try putting shorter more relevant 'hint' kind of features compared to such larger chunks of code being recommended. Best of luck with your IDE and idea. -Bye."
Good

APPENDIX B

MTURK QUALIFICATION QUESTIONS

Mechanical Turk Qualification test questions:

1. Can the same java program run on multiple operating systems without any changes in source code?

o Yes

o Maybe

o No

2. What is runtime polymorphism?

o Runtime polymorphism is a process in which a call to an overloaded method is resolved at runtime rather than at compile-time.

o Runtime polymorphism is a process in which a call to an overridden method is resolved at runtime rather than at compile-time.

o Both of the above

o None of the above

3. Select the odd one out

o Linux

o OSX

o Eclipse

o Windows

4. Based upon your prior knowledge, arrange the sorting techniques in increasing order of their time complexities (average case scenario).

Bubble Sort (B)

Insertion Sort (I)

Merge Sort (M)

Quick Sort (Q)

o B < M < Q < I

o [Q,M] < [B, I]

o [B, I] < [Q,M]

o [B, M] < [Q, I]

5. Is there a problem(s) in the below java code snippet? If yes, what?

```
public void int sum(int [] a){
    int total = 0;
    for(int i = 0, i <= a.length, i++){
```

```
        total += a[i]
        }
        return total;
}
```

o No problems

o Yes _____

6. Based upon your prior knowledge, arrange the sorting techniques in increasing order of their time complexities (average case scenario).

Bubble Sort (B)

Insertion Sort (I)

Quick Sort (Q)

Heap Sort (H)

o B < I < Q < H

o [Q, I] < [B,H]

o [B, I] < [Q,H]

o [Q, H] < [B, I]

7. Write the name of any one Wrapper class in Java

_____