2018

A Behavior-Driven Recommendation System for Stack Overflow Posts

Chase D. Greco
Virginia Commonwealth University

A BEHAVIOR-DRIVEN RECOMMENDATION SYSTEM FOR STACK OVERFLOW POSTS

A thesis submitted in partial fulfillment of the requirements for the degree of Master of Science at Virginia Commonwealth University.

by

CHASE GRECO

Bachelor of Science in Computer Science from Virginia Commonwealth University, 2017

Proposal Director: Dr. Kostadin Damevski,

Assistant Professor, Department of Computer Science

Virginia Commonwealth University

Richmond, Virginia

April, 2018

Acknowledgements

It is with the deepest feelings of gratitude and appreciation the author would like to acknowledge the following individuals for their help and support. Each of them has in some way contributed to the success of my graduate studies.

I wish to thank Dr. Kostadin Damevski, for providing the initial idea behind the tool described in this thesis, for his constant support and guidance throughout the tool's development, and his diligent reviews and comments on earlier versions of this document, for without them its quality would have surely suffered.

I would also like to acknowledge the many students who have been a part the tool's development team at one point or another, without their hard work and dedication it would not be nearly as successful, or visually appealing, as it is today.

Lastly, I would like to express my appreciation to my family for their love and support, without which I undoubtedly would not have been able to complete this journey.

TABLE OF CONTENTS

Chapter Page

LIST OF TABLES

LIST OF FIGURES

Abstract

A BEHAVIOR-DRIVEN RECOMMENDATION SYSTEM FOR STACK OVERFLOW POSTS

By Chase Greco

A thesis submitted in partial fulfillment of the requirements for the degree of Master of Science at Virginia Commonwealth University.

Virginia Commonwealth University, 2018.

Director: Dr. Kostadin Damevski,
Assistant Professor, Department of Computer Science

Developers are often tasked with maintaining complex systems. Regardless of prior experience, there will inevitably be times in which they must interact with parts of the system with which they are unfamiliar. In such cases, recommendation systems may serve as a valuable tool to assist the developer in implementing a solution.

Many recommendation systems in software engineering utilize the Stack Overflow knowledge-base as the basis of forming their recommendations. Traditionally, these systems have relied on the developer to explicitly invoke them, typically in the form of specifying a query. However, there may be cases in which the developer is in need of a recommendation but unaware that their need exists. A new class of recommendation systems deemed Behavior-Driven Recommendation Systems for Software Engineering seeks to address this issue by relying on developer behavior to determine when a recommendation is needed, and once such a determination is made, formulate a search query based on the software engineering task context.

This thesis presents one such system, StackInTheFlow, a plug-in integrating into the IntelliJ family of Java IDEs. StackInTheFlow allows the user to inter-

act with it as a traditional recommendation system, manually specifying queries and browsing returned Stack Overflow posts. However, it also provides facilities for detecting when the developer is in need of a recommendation, defined when the developer has encountered an error messages or a difficulty detection model based on indicators of developer progress is fired. Once such a determination has been made, a query formulation model constructed based on a periodic data dump of Stack Overflow posts will automatically form a query from the software engineering task context extracted from source code currently open within the IDE. StackInTheFlow also provides mechanisms to personalize, over time, the results displayed to a specific set of Stack Overflow tags based on the results previously selected by the user.

The effectiveness of these mechanisms are examined and results based the collection of anonymous user logs and a small scale study are presented. Based on the results of these evaluations, it was found that some of the queries issued by the tool are effective, however there are limitations regarding the extraction of the appropriate context of the software engineering task yet to overcome.

CHAPTER 1

INTRODUCTION

Developers are tasked with developing and maintaining ever growing and complex systems. Regardless of their prior experience, there will inevitably be times in which a developer must interact with parts of the system with which they are unfamiliar, when fixing a bug or adding a new feature. This is especially true for novice developers. In such cases, the developer must utilize additional resources to gain information about the system which they do not already possess [1]. The additional resources frequently come in the form of asking fellow teammates for assistance [2], participating in pair-programming sessions [3], or consulting the increasingly vast amount of resources available online [4].

In many cases, fellow teammates are not available to consult. To get around this, software developers frequently search for web resources in order to learn from others, and sometimes even to remind themselves of details related to development knowledge for which they are already familiar [5]. They turn to resources such as blogs, forum posts, mailing lists [6], Q&A sites, or bug-trackers [7].

Online resources have become an invaluable tool for the developer to both understand the problem they are attempting to solve and to implement a solution. Often, when developers are confronted with prototyping a new feature, particularly when functionality is more valued than stability, they utilize an "opportunistic" approach to software development. In such an approach, searching for small snippets of code to employ in a *copy-and-paste* strategy, modifying and incorporating snippets into their own code to achieve the desired functionality [8]. Similar behavior occurs when developers encounter error messages for which they are not familiar or do not comprehend, a scenario that is common among novice developers, who have difficulty interpreting

error messages [9]. As a result, many web resources have been created and developed to assist developers when encountering issues. Many of these resources come in the form of Question and Answer (Q&A) Forums, where a developer poses a question regarding some technical issue and other members of the community attempt to provide guidance or propose a solution. The Stack Overflow Q&A Forum represents one of the most prominent of these resources, its large archive of software-related posts has continued to grow in popularity with software developers, with over 40 million monthly visitors, including an estimated 16.8 million professional developers and university students [10].

The amount of resources and information available to developers online has been growing steadily over the last several years. These resources represent vast amounts of collective knowledge on topics surrounding the development and maintenance of software. Indeed, most modern software development tasks require such large amounts of information to be available and at hand so that they can be completed. The need then exists to enable developers to quickly sift through the large quantities of information available to identify the "nuggets" of information that are pertinent to solving the task at hand. One such approach to addressing this need is through a recommendation system.

Recommendation systems can aid developers in managing the large information requirements of modern software development [11]. They often utilize available online resources as a knowledge-base which is then queried to generate recommendations based on developer activity. Often, such recommendation tools are deployed within the Integrated Development Environment (IDE) itself, allowing the developer to interact with the recommendation system without the need to change context. Several recommendation tools targeting the Stack Overflow knowledge-base have been proposed with the aim of improving developer productivity by integrating relevant information from Stack Overflow into the IDE. Prompter [12] and Seahawk [13] are able to automatically recommend Stack Overflow posts based on source code context present in the IDE.

T2API [14] and NLP2Code [15] recommend code snippets extracted or adapted from Stack Overflow based on natural language text describing the programming task.

Opportunities still exist for such tools to better integrate with the IDE and to the developer's behavior, further personalizing and targeting recommendations to opportune moments in time. This thesis introduces StackInTheFlow - a tool that intends to automate the manual task of finding relevant Stack Overflow posts. StackInTheFlow is personalized to each developer and integrates closely with their IDE behavior, allowing developers to remain in a high-productivity *flow* [16]. StackInTheFlow has the following set of characteristics:

1. Automatically constructs interpretable queries based on the current source code context

2. Uses clicks on retrieved results to personalize, over time, the retrieved Stack Overflow posts to specific Stack Overflow tags

3. Automatically recommends Stack Overflow posts on compiler and runtime errors in the IDE

4. Detects when a developer is facing difficulty and not making progress and recommends Stack Overflow posts

5. Queries the Stack Overflow API (and not the periodic dump) to retrieve the most recent Stack Overflow posts

StackInTheFlow integrates as a plugin with the IntelliJ family of Java IDEs, including the popular Android Studio environment. Though the tool targets a Java IDE, the mechanisms it uses are language agnostic and can be generalized to other languages with minimal effort. This thesis includes a description of each of StackInTheFlow's features, a set of preliminary results on the effectiveness of each recommendation mechanism using field data gathered from use of the tool by developers, and developer

impressions of the tool gathered from surveys collected after attempting to solve an Android development problem.

1.1 Thesis Contributions

The major contributions of this thesis are:

- The introduction of new terminology identifying a sub-class of Recommendation Systems for Software Engineering, Behavior-Driven Recommendation Systems for Software Engineering
- The introduction of a new novel metric *Edit Ratio* for the purposes of identifying when the developer has encountered a difficulty
- The introduction and implementation of a novel difficulty detection mechanism, utilizing a three-state finite state machine
- The introduction and implementation of a novel process for extracting candidate query terms from source code and formulating queries from them, based on knowledge extracted from a larger knowledge-base
- The introduction of a novel metric *Click Frequency-Inverse Document Frequency* for personalizing the results displayed by a recommendation system to the user
- The implementation of a tool, StackInTheFlow, incorporating all of these elements

CHAPTER 2

BACKGROUND

A recommendation system [17], in its most basic form, identifies potential items of interest to a particular user and provides the user those items in the form of suggestions or recommendations. In practice, these systems are often employed in *e-commerce* domains to answer questions such as "What movie should I watch?" [18], "What books should I buy?" [19], or "What restaurant should I try?" [20]. Within the domain of software engineering, typical questions with which a recommendation may be employed to assist with may be "What software components are suitable for reuse?" [21], "How does one implement a particular interface?" [22], "How does one use a particular third-party API or library?" [23], "What documentation is relevant to this bug report?" [24], or "What code artifacts within my code base are relevant to the task I am currently working on?" [25].

This chapter introduces the concept of recommendation systems within a software engineering context. It begins with a general definition of recommendation systems for software engineering and an overview of the types of tasks they can be expected to complete and challenges faced in their development. It then examines techniques for designing these recommendation systems. Following this, a survey of contemporary recommendation tools designed to be utilized within a software engineering context is provided. The survey begins with a sampling of general-purpose software engineering recommendation tools, it then narrows its focus to software engineering tools which utilize Stack Overflow specifically.

The chapter concludes by introducing the concept of Behavior-Driven Recommendation Systems for Software Engineering, which will serve as the foundation of the recommendation system this thesis proposes. A general definition is provided as well

as an overview of the challenges faced when developing this specific class of recommendation systems.

2.1 Recommendation Systems for Software Engineering

More formally, a *Recommendation System for Software Engineering* (RSSE) can be defined as [11]:

> ... a software application that provides information items estimated to be valuable for a software engineering task in a given context.

That is, a system which focuses on providing *information* as opposed to the services provided by other software engineering tools such as build and test automation software. A RSSE also utilizes *estimation*, which distinguishes it from traditional code-search tools relying on methods such as regular expressions or call-graph visualization to extract facts. It provides items of *value*, containing novel or unexpected information, while also reinforcing topics for which the developer is already familiar. Finally, they are distinct from traditional search tools in that they emphasize providing information relevant to a specific *task* within a particular *context*.

There are a wide variety of information items and information retrieval tasks a RSSE can provide or fulfill such as:

Reusable Software Components - Recommenders can assist developers in identifying software components such as classes or methods that are suitable for reuse in other parts of an application, or that are relevant to the current task at hand.

Software Component Use Examples - There may be instances when a developer knows what software components are relevant to the task at hand but not how to properly utilize them. In such cases a recommender may provide examples of the component's use.

Code Base Navigation - A recommender may also assist the developer in navigat-

ing their own code base, for example identifying elements within the code base relevant to the task at hand.

Issue Reports - A recommender can extract relevant prior issue reports surrounding a particular task, such as a bug fix or a feature request.

Documentation - A recommendation system may be devised to provide documentation beyond usage examples relevant to a particular task, for example documentation surrounding a particular bug report, such as information regarding the change history leading up to the existence of the bug.

Expert Identification - Finally, another possible use of a RSSE is the identification of domain-area experts to recommend for a task or to offer assistance.

Despite much of the above information items being readily available within the Web or internal company systems, it can be deceivingly difficult to identify the information necessary to develop a solution to a particular software engineering issue, or to even determine that such information exists. Though many challenges faced in developing recommendation systems are shared regardless of the environment in which they are deployed, there are also several challenges specific to developing recommendation systems within the software engineering domain [26, 11]. Some issues of note include:

Data Scale - While not constrained to the software engineering domain, the sheer amount of information available from which recommendations may be drawn is constantly increasing, leading to challenges in sifting through quantity of data available to identify those small "nuggets" of information relevant to the task at hand.

Data Variety - The information which may be relevant to a particular software engineering task may be highly *heterogeneous*. Many traditional recommendation systems rely on the concept of *item* and *rank* [27]. However, no direct analog exists

within the software engineering domain. Once more, the information available may range from highly structured (code) to highly unstructured (blog posts).

Data Evolution - In other domains, such as movie or book reviews, information may have an indefinite period of relevancy. However, information within the software engineering domain evolves rapidly. Software developers have a need to understand changes within their code base and the technologies and frameworks they leverage, often multiple times a day [28]. Though not all software is so volatile, certain long-term support frameworks or libraries may remain stable for years, the fluid nature of software development is such that recommendation systems must continuously that verify the information that they present is still valid and relevant.

Data Context - Information within a software engineering task is highly context sensitive. It often holds no meaning without a grounding in the underlying process. For example, a server becoming unavailable could be the result of a software or hardware failure or due to scheduled maintenance. Without access to the underlying process which generated the information, it becomes very difficult to assign it meaning.

A successful recommendation system deployed within a software engineering domain must be able to account for these issues and ensure that they do not have an detrimental effect on the quality of recommendations generated to the point that the system is no longer useful.

2.2 Recommendation System Approaches

Once a sufficient amount of data has been collected and processed, and enough information has been collected regarding the task context, the recommendation system may utilize a recommendation algorithm to generate recommendations. These recom-

mendation algorithms come in a variety of forms [27, 11], a popular selection of which is presented below:

Collaborative Filtering - One of the most widely utilized recommendation algorithms within general domains, Collaborative Filtering [29, 30, 31] is based on the concept of "word of mouth" recommendations. The basic premise is to rely on individuals with similar interests as yours and with which you have a close relation (friends and family) to make recommendations. In its simplest form your friends and family are replaced by users with similar preferences as yours, i.e. *Nearest Neighbors*. Collaborative filtering utilizes two types of information to form its recommendations, a set of *users* and a set of *items*, with the relationship between the two typically expressed in the form of *ratings*. The basic algorithm is to then first compute users most similar to yourself based on your past ratings history, and from them extrapolate items you are most likely to also rate highly, but haven't yet encountered. Though one of the most widely utilized techniques for general recommendation, collaborative filtering is rarely utilized within a software engineering domain due to lack of direct analog of items and ratings as has been previously discussed. Instead, other recommendation algorithms have become more prominent.

Content-Based Filtering - Another popular approach, Content-Based Filtering [32, 33] utilizes the concept of categories or topics a user might be interested in, with the assumption that such topics do not drift too far day to day, and that the user will be interested in such topics again in the future. For example, a developer interested in the topic of *Java JUnit Testing* will most likely not change their interest in the topic from one day to the next, and will also be interested in the topic in the future. It utilizes two types of information to form its recommendations, a set of *users* and a set of *topics*. Recommendations are formed by first extracting topics from a set of items. Items are then recommended based on how similar they are to items that you have previously rated highly. Similarity between items is measured on the topics they share in common.

Topics can be exacted from item descriptions based on the presence of *keywords*, or in the case that items have already been pre-annotated, *categories*.

Knowledge-Based Recommendation - A different approach to constructing a recommendation system, Knowledge-Based Recommendation [34, 35, 36] does not rely on the concept of item ratings or textual descriptions, but instead on a deeper level of knowledge about the items available. Such knowledge allows for a finer-grained level of detail about an item and thus enables alternative recommendation schemes. The basic approach utilizes two types of information. The first type of information is either a set of *rules*, often called constraints, or a set of *similarity metrics*. The second type of information is a set of *items*. In such a system the similarity metric assesses how well a potential recommended item satisfies a user's need. In the case of rule-based approaches, a pre-existing knowledge-base is utilized to generate rules to determine what items are best to recommend for a particular set of user needs. Knowledge-based methods typically out-perform collaborative filtering or content-based filtering approaches initially, however if they are not equipped with a mechanism to enable learning of new information to incorporate into the model, they may be surpassed by other methods as the knowledge-base that they rely on becomes "stale".

Each of the above approaches has its own benefits and drawbacks [11]. Collaborative filtering and content-based filtering tend to be easier to set up than knowledge-based approaches as they don't required an extensive knowledge-base detailing aspects of items. They also tend to be more adaptive to new pieces of knowledge, as new ratings can be incorporated directly into the model for future recommendations, whereas knowledge-based approaches must be adapted manually unless additional learning methods are utilized. However, collaborative filtering and content-based filtering are subject to the *cold start problem* which expresses the need to provide an initial set of ratings with which to generate the model, an issue from which knowledge-based methods do not suffer. Knowledge-based methods also have the additional benefit of

being able to provide substantial explanations utilizing their knowledge-base as to why a particular recommendation was made. Finally, collaborative filtering methods, since they incorporate information from other users outside the user being recommended items, have the potential to produce new and surprising results of items that while useful to the user, have never been related to a past query triggered by their activity. A comparison between each of the approaches can be viewed in Table 1.

Table 1.: Comparison of Recommendation Algorithms

Collaborative Filtering (CF), Content-Based Filtering (CBF), Knowledge-Based Recommendation (KBR)

Quality	**CF**	**CBF**	**KBR**
easy setup	yes	yes	no
adaptable	yes	yes	no
cold start	yes	yes	no
deep explanation	no	no	yes
outside user info	yes	no	no

Finally, once recommendations have been made, considerations must be taken as to how to display recommendations to the user. In its most basic form, a recommendation system displays a ranked list of items of potential interest to the user, however other directions may be taken such as providing the user with an explanation as to why a particular item was recommended. In addition, the environment in which these recommendations are displayed is also a point of consideration. Many RSSEs seek to integrate directly into tools which developers are already familiar, namely the Integrated Development Environment (IDE) or the Internet Browser, both environments in which the developer spends a great deal of time [37]. Additional emphasis may also be placed on not only making recommendations to the user, but also explaining *why* a recommendation was made. Formulating such explanations is a non-trivial task, as in

the absence the typical recommendation concept of a rating within the software engineering domain, the method of explanation formulations must be re-examined for every new type of recommendation system developed.

2.3 Recommendation Tools in Software Engineering

Numerous recommendation tools have been proposed with the aim of assisting developers, many utilizing the approaches previously discussed. One particular class of recommendation systems is *Source Code Based Recommendation Systems* (SCBRS). The defining feature of a SCBRS is that it derives its recommendations from the source code of a software system. Due to the pervasive nature of programming within the domain of software development, such types of recommendation systems represent an important category of RSSEs. Other recommendation systems focus on instead constructing a representation of the entire context of the software development task. These *Context Representation Based Recommendation Systems* (CRBRS) attempt to utilize a selection of features, beyond just the source code, to inform their recommendations. A selection of recommendation systems from both of these approaches are presented below:

RASCAL - A SCBRS, RASCAL [21] seeks to recommend the next method a developer might utilize. It does this by analyzing classes similar to the one the developer is currently working on. RASCAL utilizes a collaborative filtering approach to generate its recommendations. However, unlike the traditional collaborative filtering scheme, *users* in this case are classes and *items* are methods. The similarity between the current active class and other classes is computed based on the frequency of the methods they call.

RASCAL consists of four components, by which it formulates its recommendations. The *Active User* defines the current class that the developer is working on. The *Usage History Collector* automatically mines the frequency of method calls within a class

for all classes within a given set of APIs, and the order in which they are called. The *Code Repository* stores the information mined by the history collector. Finally, the *Recommender Agent* recommends the next method a developer should consider utilizing in the implementation of the active class and current method at the position of the cursor. The agent works by computing the similarity between the current active class and other classes within the code repository. Similarity is calculated between two classes by comparing the frequency of the methods they call. Commonly occurring methods such as `toString()` are penalized in the similarity calculation. The most similar class is selected and its methods examined to form recommendations. Methods are suggested in the order in which they occurred in the most similar class, after the call to the current method.

Hipikat - Another SCBRS, Hipikat [38] takes a much broader view, beyond methods, in the types of information it recommends. It seeks to assist new developers on a project find information relevant to the development task at hand. This information can come in the form of code examples, API documentation, or other forms of electronic communication such as emails, bug reports, or forum posts. In order to utilize Hipikat, the developer views an artifact in question within the IDE (such as a bug report) and selects the "Query Hipikat" option from the tools menu. Hipikat then retrieves relevant artifacts and displays them in a separate window. For each recommended artifact, Hipikat also displays its name, type (website, news article, CVS revision, bug report), explanation as to why it was recommended, and a relevance estimate to the current artifact. From this view the developer can proceed to investigate recommended artifacts further, or use them as the basis for additional Hipikat searches.

Hipikat consists of two components. The first is an Eclipse plug-in which sends search queries, consisting of artifacts of interest or keywords, and displays retrieved relevant artifacts. The second is a back-end consisting of a relationship graph between available software artifacts, and a system to determine what artifacts are relevant to a

particular query. Unlike RASCAL, recommendations are based on the links between different artifacts in addition to their similarities. Links are established based on novel heuristics such as bug report IDs being matched to change logs, or the closing time-stamp of bug reports being matched to the revisions of source code files.

Mylyn - Unlike the previous two described systems, which relied primarily on source code as the basis of recommendation formulation, Mylyn [25] seeks to build a more complete representation of the context surrounding a software development task with which to make recommendations, as such it is a CRBRS. It does this by constructing a model encapsulating the context surrounding the software engineering task. Utilizing this model the "degree of interest" the developer may have for each class within the project is calculated, with classes with high interest being prominently displayed, while classes with low interest fading to the background.

Mylyn consists of two components, the first is an Eclipse plug-in which displays the current active task and emphasizes or hides various project artifacts based on their computed relevance to the current task. The second is a *Task Context* model, which is responsible for maintaining a relevancy score, termed "degree of interest" (DOI) for every class in the project. Mylyn is based on the intuition that for a given task not all classes within a large project will be relevant, thus classes deemed to have a low DOI by the model should not be displayed to the user for consideration. The model works by tracking developer interactions with files during the completion of a task, files for which the developer interacts with frequently during the task are judged to have high DOI and thus are displayed more prominently. Consequently, files for which the developer did not interact are filtered away. A separate model is maintained for each software development task for which the developer is assigned.

2.4 Stack Overflow Recommendation Systems

In addition the recommendation systems discussed in Section 2.3 which focused on either utilizing external information from open-source projects or the internal source code of the project itself to make recommendations, several recommendation systems have also be proposed to leverage the multitude of knowledge available within Stack Overflow. Several of the tools rely on a data dump of articles periodically released by Stack Overflow to train their recommendation models. A selection of recommendation systems utilizing this approach are presented below:

Example Overflow - A code snippet recommendation system, Example Overflow [39] seeks to recommend relevant code snippets extracted from Stack Overflow within a particular domain. Example Overflow currently targets the *JQuery* domain of software engineering tasks, however the approach is extendable to other domains as well. It works by first constructing a database of code snippets extracted from Stack Overflow within the target domain. This database can then be queried with natural language similar to a Google search. When a query is executed, the top 5 relevant code snippets are displayed side by side to enable easy comparisons by the developer.

Example Overflow consists of two components, the first is a website which enables users to enter queries and browse results. The second is a database of code snippets. This database is constructed utilizing the Stack Overflow API, articles are extracted tagged with "JQuery", articles not containing code snippets are filtered out. These articles and corresponded code snippets are stored within a database. This database is then indexed with Apache Lucene [40], which utilizes a term frequency-inverse document frequency (tf-idf) based approach to index documents. Each code snippet is treated as its own document for the purposes of document indexing and searching. Once indexed, this database can then be queried utilizing natural text and relevant code snippets retrieved.

SeaHawk - In contrast to the previously mentioned recommendation systems, in-

stead of recommending code snippets directly, SeaHawk [13] seeks to enable the developer to browse Stack Overflow as a whole within the IDE. It does this by allowing users to search for and browse entire articles extracted from the Stack Overflow Data Dump through a user interface provided within the IDE itself.

SeaHawk consists of two components, the first is an Eclipse plug-in providing the user interface to enable searching and browsing articles. This user interface is primarily composed of three elements. The *Document Navigator View* allows the user to specify search queries and browse related documents presented in the form of a tree. Users can also tag articles as relevant to the currently open source code file to be retrieved for later use within the view. The *Suggested Documents View*, presents a selection of documents similar to the navigator view that are deemed relevant to the currently open source code file. Finally, the *Document Contents View* allows the user to browse a selected article in its entirety in the form of a web page displayed within an in-IDE web browser widget. The second component of SeaHawk is a database containing all of the articles within the Stack Overflow Data Dump. This database is then indexed with Apache Solr, which utilizes a tf-idf approach to retrieve relevant documents, to enable searching for articles with natural language. Seahawk also provides a mechanism to automatically extract queries from source code to be utilized in a search. The main drawback of this approach is that the database must be periodically updated with new articles, otherwise the information it contains may become "stale" and no longer relevant to current software engineering tasks.

NLP2Code - An unconventional take on the traditional recommendation system model, NLP2Code [15] seeks to provide developers with relevant code snippets given a query such as "Split string by...?". It does this by utilizing an inventory of software development task descriptions mined from the Stack Overflow Data Dump. When the user begins entering a query, a selection of completed queries is suggested to the user from the task inventory. Once a given query is made, a Google search of Stack

Overflow articles tagged with "Java" is performed. Code snippets are extracted from the top answers of the first three retrieved articles and are suggested to the user. Unlike other recommendation systems, no additional views are utilized to enable the user to enter queries or display results, instead all interaction is done in the the source code editor of the IDE itself, with users directly entering queries within the source code.

NLP2Code consists of two components, the first is an Eclipse plug-in which enables users to enter the queries within the source code editor, as well as utilizing those queries to perform custom Google searches to extract and then display relevant code snippets. The second is an inventory of task descriptions mined from the Stack Overflow Data Dump utilizing the TaskNav algorithm [41]. This allows NLP2Code to suggest logical completions to queries such as "Split string by...?", for example "Split string by character?". In addition, the utilization of a task inventory allows accounting for the same task being expressed in multiple different forms, such as active or passive voice. It also allows distinguishing between tasks such as *convert String to int* versus *convert int to String*, something traditional "bag of words" approaches cannot handle. Once a task to be queried has been identified, relevant Stack Overflow articles can be retrieved via Google and code snippets extracted and recommended to the user.

2.5 Behavior-Driven Recommendation Systems for Software Engineering

The recommendation systems discussed up until this point have all relied on the user to prompt the system for input, whether that be entering a query into the system directly, or by prompting the system to search on their behalf. Such systems are useful, but not without limitations, for example there may be instances where the developer is in need of a suggestion but is unaware that their need exists. Such situations are similar to the interactions between two programmers participating in pair programming, working together to develop software. Each developer has their own role, the *driver* who is responsible for writing code, and the *observer*, who observes the work of the driver

[42]. It is the objective of the observer to understand the context in which the software is being developed and, when confident, interrupt the driver with suggestions. In addition, if the driver is unconfident, she may consult with the observer for suggestions. It is this type of interaction, that of an observer interrupting the developer with suggestions and being available for consultation when needed, that a *Behavior-Driven Recommendation System for Software Engineering* (BDRSSE) attempts to encapsulate.

More formally a BDRSSE can be defined as: A recommendation system for software engineering that utilizes developer behavior to inform when recommendations should be made without any direct input from the developer. That is, a recommendation system which passively observes developer activity and, when confident that the developer would benefit from a recommendation, interrupts the developer to provide one, much like the observer role within pair programming. Such a system has the potential to provide great benefit, as the timely discovery of a critical piece of information can have a dramatic impact on developer productivity [43]. However, great care must be taken when designing such systems, as interrupting the programmer at an inopportune time can have just as dramatic impediment on productivity [44]. Though this can be reduced by presenting recommendations in an unobtrusive way.

There are two sub-problems then that must be considered by BDRSSE specifically. The first is determining when the developer is in need of a recommendation. Once possible method of addressing this issue, particularly within the context of a recommendation system leveraging Stack Overflow is determining when the developer has encountered some form of difficulty, with the intuition being that in such situations a recommendation has greatest potential for a positive impact. The second problem is once the opportune moment to make a recommendation has been detected, forming the query to search for relevant information items without the developer directly specifying it. A selection of approaches to both problems are presented in the following sections.

2.5.1 Difficulty Detection

The first problem to be addressed by a BDRSSE is that of identifying when the developer is in need of a recommendation. Such a situation is not well defined however, one method of circumventing this issue is to re-frame the problem to that of detecting developer difficulty. More formally, identifying instances when the developer has entered a situation or mental state which impedes development. Several systems have been proposed to address this issue, however many rely on bio-markers which are obtrusive to measure such as skin conductivity [45], or pupil dilation and heart rate [46]. While such systems do show promise, they are not practical to deploy beyond an academic setting. An alternative approach is to instead infer the developer has encountered difficulty based on logs collected of their interactions with some component(s) of the system. Such approaches also illustrate some promise. Fogarty et al. [47] have shown that it's possible to develop a system that utilizes developer interactions within the programming environment to determine if they are in an interruptible state.

There are several approaches to utilizing development logs to detect developer difficulty. The simplest and most intuitive of these approaches is to log indicators of developer progress, such as the writing of new lines of code, or the creation of new classes, and listen for periods of time where such indicators do not occur. While intuitive, this approach has several drawbacks, namely if the developer has simply gotten up to get a cup of coffee, the system will incorrectly identify the developer as encountering difficulty and begin making recommendations.

An extension of the previous approach is to take into account the frequency of "progress" events within a given time interval and utilize thresholds below which the developer will be classified as encountering difficulty. However, in addition to the drawbacks of the base approach, Nair and Mynat [48] found that in a similar task of identifying developer task switches, such frequencies are highly dependent on the individual developer, further complicating developing an accurate model.

A third approach, proposed by Carter and Dewan [49], utilizes the ratio between events deemed to be making progress such as editing events to those deemed to be indicators of difficulty such as navigating or the development environment losing focus. If the ratio of progress events to non-progress events within a fixed window of the last n events to occur falls below a certain threshold, the developer is classified as encountering difficulty. Such an approach does not suffer from issues surrounding periods where there developer is not directly interacting with the system, an also allows for personalization to the individual by fine-tuning ratio thresholds based on past developer activity [50].

2.5.2 Automatic Query Formulation

The second issue which must be addressed by a BDRSSE is that of query formulation. Once it has been determined that the developer would benefit from a recommendation a query must be formulated to search a collection of potential knowledge items to recommend. Once more, this query must be formulated without any direct specification from the user. While there are several methods that may accomplish this, one popular approach is to extract query terms from the source code of the currently open file within the IDE. When considering potential query terms, often the saliency or information-content of a term is utilized within a weighting scheme to extract candidate query terms, with the intuition being the most information rich terms would form the most beneficial query. When assessing the saliency of terms within a document such as a source code file, there are three general types of features that are often employed: Lexical, Syntactic, and Semantic.

Lexical - The simplest types of features, lexical features consider themselves with the term itself. In traditional natural language processing, these could be the frequency of the term in a given phrase, or its lemmatized form. For the process of query formulation within a BDRSSE, one simple approach utilizing lexical information would be given a source code file, compute the frequency of every term in the file, then select

the n most frequent terms to form the query. Such a procedure is often deemed a "bag of words" approach and is often combined with a list of stop-words which are automatically removed from consideration, in general English these could be terms such as "it" or "the" which though they may occur frequently do not provide any meaningful information. The development of stop-words lists for source code is an interesting topic for future research. The main issue with utilizing such features within a software engineering domain is that the most frequently occurring terms may not be the most discriminative in selecting terms that appropriately describe the current context. For example, terms like *print*, *import*, or *catch* may occur very frequently within source code and wouldn't be considered stop-words in general English, however have too generalized a meaning in software to adequately describe a context.

Syntactic - Syntactic features consider themselves with how terms fit together. In traditional natural language processing, this could be the part of speech of a term. Within software engineering, this could be identifying phrases such as class or method declarations, import statements, or loop structures. When considering extracting syntactic features for the purpose of query formulation within a BDRSSE, it is generally considered too expensive to construct the entire abstract syntax tree of a source code file so instead an "island parsing" approach is utilized to extract only the structures of interest from the file. Syntactic features have been successfully utilized to create query formulation methods within RSSEs [51] and are often used to augment lexical features in practice.

Semantic - The most complex types of features, semantic features consider themselves with the *meaning* behind a word and typically require some form of external knowledge-base from which that meaning is derived. In traditional natural language processing, these features are often related to the definition of the particular sense of the word that is in use. Within the domain of software engineering, such features are typically not utilized when constructing query formulation methods, though some work

has been done examining the entropy [52] of terms within a knowledge-base such as Stack Overflow as a possible method of term selection [13].

2.5.3 Prompter

To the author's knowledge, the only BDRSSE targeting Stack Overflow which has been proposed is PROMPTER [12]. PROMPTER seeks to fulfill the role of a prompter in a theater, "ready to provide suggestions whenever the actor needs them, and ready to autonomously give suggestions if it feels something is going wrong".

PROMPTER consists of three components. The *Eclipse plug-in* provides the user interface by which the developer interacts with the tool within the IDE. The *Query Generation Service* is responsible for extracting queries from the context of the software development task. The *Search Service* is responsible for passing generated queries off to a multitude of search engines and retrieving their results. A description of each component is provided below:

The *Eclipse plug-in* provides the user interface of the tool as well as task context tracking capabilities. The user interface consists of a notification center which displays the last ten notifications made by the tool. Each notification contains a confidence score of how certain PROMPTER is that the linked Stack Overflow article is relevant to the current task. The developer can fine-tune a confidence threshold under which notifications will not be made utilizing the notification center as well. When a notification is selected, a document view is opened allowing the developer to browse the entire contents of the article as well as rate its relevancy. The second function of the Eclipse plug-in is to track the context of the current software development task. It does this by listening to the developer's edit actions, when the developer stops writing, the plug-in identifies the current program element (method or class) and extracts the task context which consists of (i) the fully qualified package name identifying the element, (ii) the source code of the modified element, (iii) the types used of any external API, and (iv)

the names of any methods called from an external API. This information is then sent to the query generation service.

The *Query Generation Service* provides a means of extracting a query given a software engineering task context. It does this by utilizing the terms extracted by the Eclipse plug-in and computing their entropy within the Stack Overflow Data Dump. Terms with lower entropy are deemed to have higher information content. The service ranks potential terms based on a term quality index (TQI) defined by:

$$TQI_t = v_t \cdot (1 - E_t) \tag{2.1}$$

Where v_t is the frequency of a given term t and E_t is its calculated entropy. Ponzanelli et al. find that this metric works quite well, but has one drawback, which occurs when misspellings are scored as candidate query terms. Misspellings are rated as having low entropy due to their infrequent occurrence within the Stack Overflow Data Dump. To circumvent this issue, the *Levenshtein distance* [53] is calculated between terms within a context to identify and discard potential misspellings. Once a query is formed is it sent to the Search Service.

The *Search Service* sends a generated query to several search engines (e.g. Google, Bing, Blekko) to retrieve Stack Overflow articles. The articles are then ranked across several metrics, textual similarity to the query, similarity between the article and source code, similarity between API types and methods, the question and accepted answer scores, the reputation of the posting user, and the similarity between the article's tags and source code import statements. From the scores across each dimension the articles are ranked an and confidence is computed. Articles with a high confidence are then recommended to the user.

The recommendation system proposed by this thesis, StackInTheFlow, is a BDRSSE like Prompter, however it differs in several key areas:

1. Firstly, the user interface of StackInTheFlow mimics that of a more conven-

tional search engine. Importantly, the search bar is always present, even for automatically generated queries, meaning the user is able to modify and adapt these queries as they see fit.

2. In addition, a more sophisticated difficulty detection model is utilized to determine when the developer is in need of a recommendation, rather than waiting until the developer has stopped editing.

3. Unlike Prompter, which relies on external search engines to perform queries for Stack Overflow articles, StackInTheFlow utilizes the Stack Overflow API to query the Stack Overflow website directly.

4. Finally, StackInTheFlow attempts to personalize the ranking of the recommended articles based on the user's past interactions with the tool.

A full description of StackInTheFlow, including a detailed explanation of these various mechanisms can be found in Chapter 3.

CHAPTER 3

A BEHAVIOR-DRIVEN RECOMMENDATION SYSTEM

This chapter introduces StackInTheFlow - a Behavior-Driven Recommendation System for Software Engineering (BDRSSE), a recommendation system which utilizes the developer's behavior to inform when the recommendation should be made. Its objective is to automate the manual task of finding relevant Stack Overflow posts. StackInTheFlow is personalized to each developer and integrates closely with their IDE behavior, allowing developers to remain in a high-productivity *flow* [16]. StackInTheFlow integrates as a plug-in within the IntelliJ family of Java IDEs, including the popular Android Studio environment. Though the tool targets a Java IDE, the mechanisms it uses are language agnostic and can be generalized to other languages with minimal effort. For languages which utilize Java-style import statements such as Scala, the tool can be utilized without modification. This chapter begins with an overview of StackInTheFlow, and how it addresses the issues faced by BDRSSEs. It then gives description of the user interface and use-cases of StackInTheFlow. Finally, a description of the tool architecture, and overview of the models and techniques utilized within the recommendation system are presented.

3.1 Overview of StackInTheFlow

StackInTheFlow is a BDRSSE that seeks to recommend relevant Stack Overflow articles to the task at hand without any direct input from the user, particularly when the user has encountered some form of difficulty. It does this by utilizing events extracted from development logs of the user activity, specifically the presence of compile or runtime error messages, and the editing progress of the user within the IDE. Utilizing these data points, a model is developed utilizing a similar method to that of

Carter and Dewan [49], that of heuristically setting event ratio thresholds to determine when a user has encountered difficulty and thus form a recommendation. Recommendations are generated based on a separate model generated from the Stack Overflow Data Dump. In addition to this automated approach, StackInTheFlow also allows the user to interact with it as a traditional RSSE, manually entering queries and browsing search results. Finally, a hybrid approach may be used in which the user explicitly invokes the automated article retrieval process. This allows StackInTheFlow to be more flexible in the roles it can fulfill and for a greater variety of tool use-cases. For a full description of use-cases see Section 3.3. StackInTheFlow also has the capability of personalizing the ranking of results to the context of individual users via a process described in Section 3.4.4.

There are also several challenges faced when developing a BDRSSE, both those faced by all RSSEs and those faced by a BDRSSE in particular. An overview of the challenges faced by each class of recommendation system may be found in Chapter 2. The following subsections describe how StackInTheFlow attempts to address each of these issues.

3.1.1 Addressing the Challenges of RSSEs

As identified in Chapter 2, there are several challenges faced with developing RSSEs. Theses are the challenges of *Data Scale*, *Data Variety*, *Data Evolution* and *Data Context*. Without addressing these challenges, a RSSE cannot be successful. An overview of how StackInTheFlow addresses each of these issues is presented below:

Data Scale - Data scale refers to the vast amounts of information available, from which a RSSE can draw to form its recommendations. A traditional recommendation system must concern itself with indexing and organizing this information in such a way that it can be easily retrieved as the result of a query. To circumvent this issue StackInTheFlow utilizes the Stack Overflow API which provides

mechanisms to retrieve relevant Stack Overflow articles based on a query. In this way, the burden of constructing and storing an index of documents no longer rests with the tool, but on the Stack Overflow API itself.

Data Variety - Data variety refers to the heterogeneous nature of the information which may be relevant to a software engineering task. A RSSE which utilizes multiple sources of information must be able to extract information from each source and present it in a meaningful way, which is logical from the perspective of the user. StackInTheFlow chooses to focus on one form of information, the Stack Overflow article, to eliminate some of the concerns which arise from recommending multiple forms of information.

Data Evolution - Data evolution refers to the short shelf life of information within the software engineering domain. A successful RSSE must be able to constantly update the information it recommends to ensure it stays fresh. Many recommendation systems [13, 39] rely on drawing recommendations from a static knowledge-base, for example the Stack Overflow data dump. In contrast to this approach, StackInTheFlow utilizes the Stack Overflow API to fetch articles from the live Stack Overflow website, ensuring that users are always recommended the most up-to-date articles. Such an approach eliminates the need to constantly update a knowledge-base to ensure recommendations remain fresh.

Data Context - Data context refers to the highly context-sensitive nature of information within the software engineering domain. Information often holds no meaning without an underlying knowledge of the context from which it was generated. To accommodate this StackInTheFlow utilizes a content-based filtering approach to keep track of the *tags* associated with presented Stack Overflow articles which are relevant to the current context. A full description of this mechanism can be found in Section 3.4.4.

3.1.2 Addressing the Challenges of BDRSSEs

In addition to the general issues a RSSE must address, Chapter 2 also identified two problems faced by BDRSSEs specifically, that of identifying when a recommendation should be made and, once such a determination has been made, how a query should be formulated to retrieve knowledge items to recommend. A brief description of how StackInTheFlow addresses both of these issues is given below.

Recommendation Determination - The first problem that must be addressed by a BDRSSE is identifying when a recommendation should be made to the user. One method of addressing this problem is to re-frame it as detecting when the user has encountered some form of difficulty as it is an easier situation to define. StackInTheFlow utilizes this approach, seeking to make recommendations when it has determined the developer has encountered some form of difficulty. It does this via a two-pronged approach. The first approach listens for error messages as an indicator of developer difficulty and makes a recommendation when such a message occurs. The second approach utilizes a method similar to that of Carter and Dewan [49], setting ratio thresholds between events deemed to be indicators of progress and indicators of difficulty. When the ratio between difficulty events and progress events crosses a certain threshold, a recommendation is triggered. A full description of this mechanism can be found in Section 3.4.1.

Query Formulation - Once it has been determined that a recommendation should be made, the next problem that must be addressed is that of formulating a query to retrieve relevant items to recommend. StackInTheFlow attempts to solve this issue by utilizing a combination of syntactic and semantic features (see Chapter 2 for a description of query formulation feature types). Firstly, potential query terms are extracted from the bodies of posts contained within the Stack Overflow Data Dump, these terms are scored on a variety of query quality metrics such as *term frequency-inverse document frequency* to form a static dictionary of terms. When a query is to

be formed, terms are extracted from the `import` statements and selected line of the currently open document in the editor. Theses terms are scored based on the metrics contained within the previously generated dictionary. The highest scoring terms are then selected to form the query. A full description of this process can be found in Section 3.4.2.

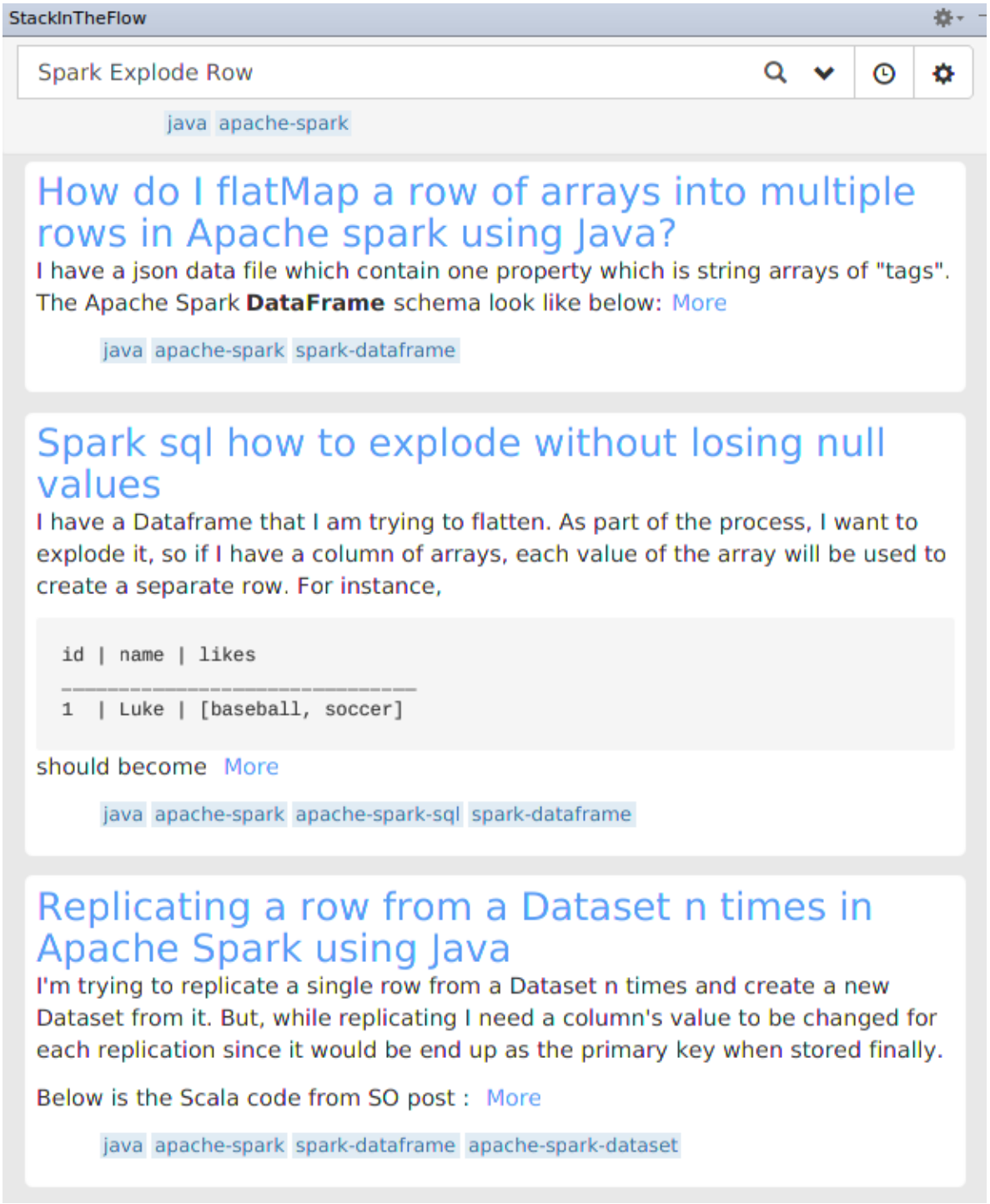
With each of the above problems addressed, StackInTheFlow can be deployed as a BDRSSE. The remainder of this chapter is structured as follows. Firstly, a description of the user interface of StackInTheFlow is given. Second, an overview of the different use-cases in which the developer can interact with StackInTheFlow is provided. Finally, the architecture of StackInTheFlow is detailed, including the mechanisms for difficulty detection and query formulation described above.

3.2 User Interface

Figure 1 shows the user interface of StackInTheFlow. It provides a tool window which can be positioned by the user within the IDE. This tool window provides a query box, much like a standard search engine, in which users can enter queries in natural language, such as "Spark Explode Row" for searching for information about the *explode* operation within the Apache Spark Framework.

The main area of the tool window displays the results of a query in the form of question posts retrieved directly from Stack Overflow through their API. For each retrieved question post the title is rendered along with a snippet of the text from the question body. Any tags associated with the question are included at the bottom of its entry as well. The question title serves as a link to the question post on Stack Overflow itself. Selecting the title will open the full post in the user's preferred Internet browser application. The question body is rendered such that code snippets are formatted in mono-spaced font, and that formatting included by the user such as bold and italics are properly displayed. Links included in the question body are also correctly formatted

Figure 1: StackInTheFlow User Interface.



and will direct the user to the webpage to which they link in the user's preferred browser when selected as well. Users may "expand" the snippet of text from the question body displayed by selecting the `more` link. Doing so will display the body of question text in its entirety along with a `less` link by which the user can revert the amount of displayed text to its previous state when selected. In addition, any tags which have been assigned to the post are displayed at the bottom of its entry in the result list. These tags may be selected by the user to serve as filter criterion to further refine their search.

Directly below the query box, a list of active tags may optionally be displayed, these tags can be enabled by selecting them from question descriptions as previously described or through the advanced query syntax detailed in Section 3.2.1. Once enabled, only questions tagged with all of the enabled tags will be displayed, however questions may include tags beyond those that are enabled. In this way, the user can limit their search results to posts tagged with topics that are of interest such as `java` or `apache-spark`. To disable a tag, the user may simply select it from the list of active tags, at which point it will be removed and the search results updated to reflect the new filter criteria.

StackInTheFlow also comes with the standard result ranking options one would expect, which are available via a drop-down menu to the right of the query box. Users can re-rank results based on the number of *votes* the post received from the Stack Overflow community, the *newest* post to be made, the timestamp of the last *activity* (initial post or reply) made for a post, or the *relevancy* of the post to the query. Each of these rankings are determined by an unspecified algorithm within the Stack Overflow API, however results ranked utilizing the *relevancy* option are further re-ranked by StackInTheFlow utilizing a procedure detailed in Section 3.4.4.

Users may also view their past queries via the *history* tab, denoted as a clock. Users may select a past query to execute it again. In addition to the query string itself, the associated tags utilized to filter a past query are stored and displayed as well.

Finally, users may access various options within the *settings* tab, denoted as a gear.

Importantly, users may enable or disable under what conditions StackInTheFlow will produce an automated recommendation. Specifically, users may control whether StackInTheFlow will issue automated queries upon compile or runtime errors, or when the tool has detected the user has encountered difficulty. Details of each type of query scenario can be found in Section 3.3.

3.2.1 Advanced Query Syntax

In addition to basic natural language query strings such as "Spark Explode Row", StackInTheFlow also supports a more advance query syntax by which users can formulate queries. This query syntax is provided by the Stack Overflow API. Of note, users can specify specific tags to be include in their search by enclosing them in square brackets (e.g. "[java]") an operation that is equivalent to adding the tag to the list of active tags. Users can also search specific elements of a question by specifying the element followed by a colon and the string to search for (e.g. "title:"Spark"'"). A full listing of the supported query operations can be found within the Stack Overflow Search Syntax Specification[1]

Finally, due to the inclusion of tags being a common user operation, StackInTheFlow provides a shortcut via the `Tab` key, by which users can quickly add tags. When entering a query, users can press the `Tab` key to automatically add the previously typed word to the list of active tags.

3.3 Tool Use-Cases

As previously described, the first issue a BDRSSE must address is that of determining that the user is in need of a recommendation. This can happen in two ways: 1. The developer may manually invoke the tool. In this case the tool is acting as a conventional RSSE. 2. The tool may utilize the developers behavior to determine that

[1]Stack Overflow Search Syntax Specification https://stackoverflow.com/help/searching

a recommendation should be made. In this cases the tool is fulfilling the role of a BDRSSE. StackInTheFlow seeks to account for both avenues by providing functionality to enable use cases from either path. These use-cases can be described by the type of query that initiates them: *manual*, *automatic*, *error*, and *difficulty*. Manual queries allow the developer to utilize StackInTheFlow as a conventional RSSE, manually specifying the query string. Automatic queries are designed to be a hybrid approach where the developer explicitly invokes the automated query generation procedures of the tool. Error and difficulty queries represent the behavior-driven aspect StackInTheFlow, invoking the automatic generation of queries whenever the user has encountered a perceived error or difficulty. To illustrate each use-case, consider a scenario where a developer has just begun using the Apache Spark parallel programming framework.

3.3.1 Manual Query

The developer may utilize StackInTheFlow as a conventional RSSE by manually writing and issuing queries to the Stack Overflow API, such as a query regarding the Spark *explode* operation (as in Figure 1). From there she may browse the results of her search directly within the IDE through the user interface, allowing for all of the operations previously described in Section 3.2. Importantly, this use-case mirrors the functionality of the Stack Overflow website itself, providing a method for users to interact with StackInTheFlow which which they are most likely already familiar. In addition, this method can be utilized to modify the queries generated by the use-cases described in the following sections, allowing the user to further personalize and fine-tune the results generated by more automated methods.

3.3.2 Automatic Query

Occasionally, the developer may not be able to form a query suitable to retrieve the information required to solve a development problem and may require assistance in query composition. For example, the developer may wish to know how to set the configuration options for the *SparkSession* object. In such a case she may simply highlight the section of code relevant to declaring or using this object, right-click and select the *Auto Query* option. Utilizing the procedure detailed in Section 3.4.2, StackInTheFlow will automatically generate a query from the code snippet and present the results in the same fashion as above. This use-case represents a hybrid approach, incorporating the automatic query generation aspects of a BDRSE with the explicitly invoked nature of a conventional RSSE.

3.3.3 Error Query

Inevitably, during the course of their daily work a developer will encounter error messages. These messages can often be cryptic and unfamiliar to the developer, requiring the consultation of sources such as Stack Overflow in order to decipher their meaning. To address this issue, whenever an error message is encountered, either during compile or run time, StackInTheFlow will generate a query and recommend results using the approach described in Section 3.4. This use-case represents a simple approach to user difficulty detection, the intuition being that error messages serve as an indicator of developer difficulty. A more advanced method of user difficulty detection is utilized in the following use-case.

3.3.4 Difficulty Query

Finally, it may be the case that the developer is stuck in an unproductive loop. She may be deleting large portions of code without making significant progress, or scrolling through files without making any edits. StackInTheFlow contains mechanisms to

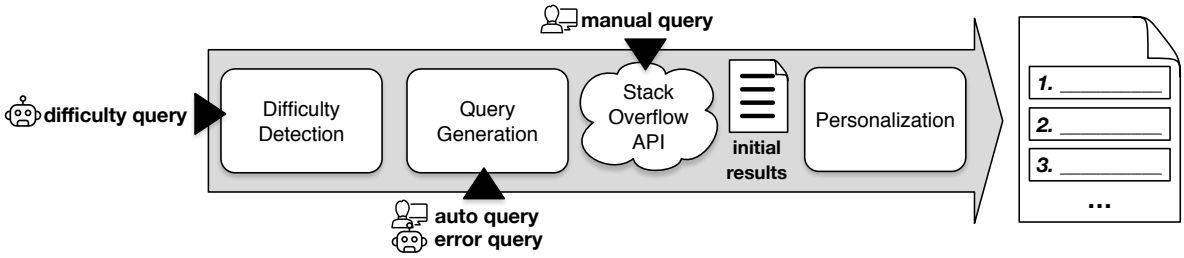
detect such behaviors, outlined in Section 3.4, and to automatically generate queries and present recommendations without any direct input from the developer, which may provide the information they need to overcome their development block. This use-case represents the fully behavior-driven aspect of StackInTheFlow, qualifying it to be deemed a BDRSSE.

3.4 Tool Architecture

Unlike other RSSEs, StackInTheFlow does not store a repository of knowledge items (articles) to recommend. It instead utilizes the Stack Overflow API to dynamically query the live Stack Overflow website. This was done to ensure that the user would always have access to the most up-to-date articles. As a consequence of this approach, much of the burden typically associated with developing a RSSE (storing and indexing knowledge items, determining what items are relevant to a particular query, etc.) is instead offloaded to the Stack Overflow API itself, this means that StackInTheFlow is a much lighter weight application than many of its counterparts and its architecture reflects such. The major drawback of this approach however is that in order to utilize the tool, the developer must have access to a reliable Internet connection, a barrier that was deemed to be acceptable, given it is often seen as a prerequisite within a software development environment.

The different components of the StackInTheFlow architecture and their relationship to the different use-case queries is shown in Figure 2. Depending on the use-case, the procedure by which StackInTheFlow generates a query is slightly different. Subsequent sections will describe the internals of each component and how each use-case procedure is performed in detail.

Figure 2: Overview of StackInTheFlow



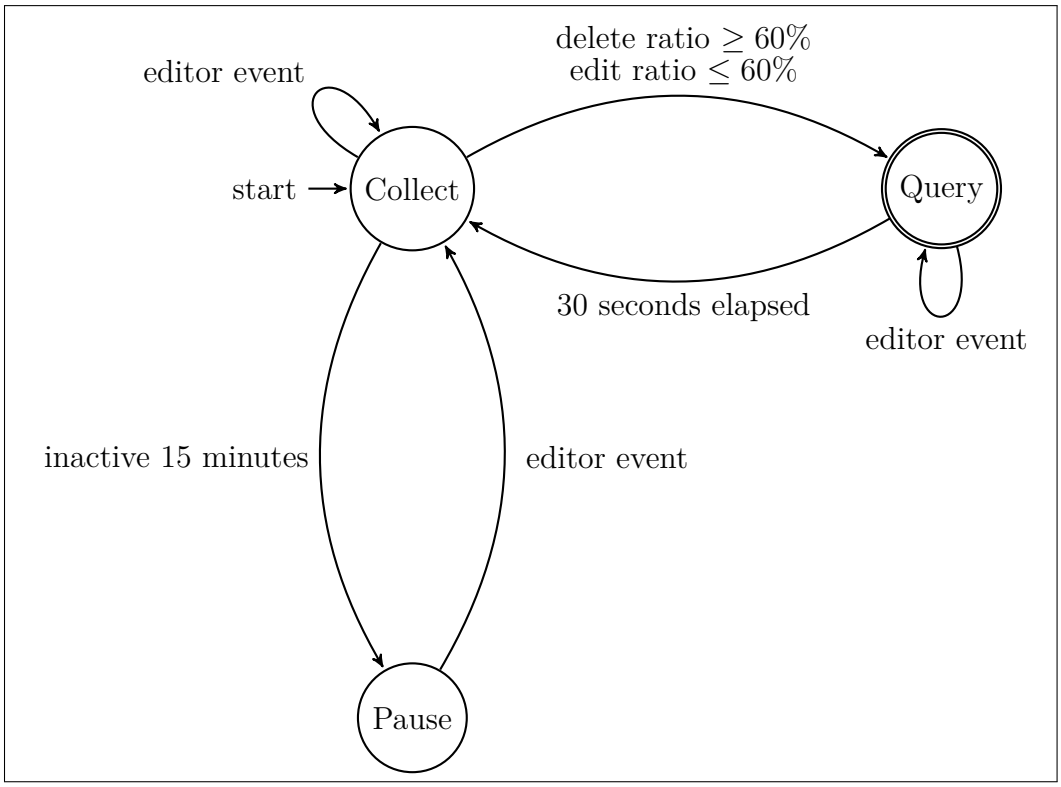
3.4.1 Difficulty Detection

The first component of StackInTheFlow is that of the Difficulty Detection module. This module is designed to address the first issue faced by a BDRSSE, that of determining when the developer is in need of a recommendation, and therefore is also responsible for enabling the difficulty query use-case. To do so it must be able to detect when the user has encountered some form of "difficulty", at which point it triggers the subsequent modules in the chain to produce a recommendation. One such perhaps obvious case is when compiler or runtime error messages are encountered. Such a situation may be seen as a clear indicator that the developer has encountered some form of difficulty. StackInTheFlow employs this viewpoint, as such whenever an error message is encountered, be it compile or runtime it is immediately sent to the Query Generation module for further processing. However, the determination of programmer difficulty is not always such a trivial task.

In the case of an error message, the indication of the user encountering an obstacle is quite clear, however, the user may encounter difficulty without as prominent an indicator. They may be constantly editing the same section of code, or unable to continue development due to lack of project understanding. StackInTheFlow provides mechanisms to detect such cases and to provide Stack Overflow posts to aid the developer, by leveraging its automatic query generation capability. It does this by taking an approach similar to that of Carter and Dewan [49], by analyzing the ratios between the

insertion and deletion of text within the editor. Such an approach, Carter and Dewan argue, prevents the tool from falling into the trap of determining the programmer has encountered difficulty when they have simply *"gotten up to get a cup of coffee"*. This approach utilizes the finite state machine detailed in Figure 3.

Figure 3: Difficulty Detection State Machine.



The machine begins in the *Collect* state. While in this state, editor events are collected. Editor events are broken into four categories: *Insert*, *Delete*, *Scroll*, and *Click*. *Insert* and *Delete* events are fired when the user inserts or deletes a character respectively. *Scroll* events are fired when the user scrolls the view within the editor. *Click* events are fired when the user clicks the mouse. A queue of the past 25 events is maintained. In order to control for event bursts, consecutive events of the same type within one second are ignored and not added to the queue except for the initial event. In this way, event categories which trigger multiple consecutive events, such as scrolling, do not overpower and erase event categories which produce sparser event sequences such

as mouse clicks. A window size of 25 was selected heuristically, however future work may examine the effect of varying window sizes. Whenever an event is added to the queue the metrics of the *delete ratio* and *edit ratio* are re-calculated, defined as:

Let H be the set of the last 25 editor events included within the history queue.
Let I be the subset of H such that for every event e, $e \in H$, e is an Insert Event.
Let D be the subset of H such that for every event e, $e \in H$, e is a Delete Event.

Let $|X|$ denote the cardinality of set X.

The **Delete Ratio** is defined as:

$$\frac{|D|}{|I \cup D|} \tag{3.1}$$

That is, the ratio between the number of deletion events, to the sum of deletion and insertion events within the past 25 editor events within the history queue.

The **Edit Ratio** is defined as:

$$\frac{|I \cup D|}{|H|} \tag{3.2}$$

That is, the ratio between the number of edit events (insertion and deletion), to that of all events, including scroll and click events within the past 25 editor events within the history queue.

Utilizing these metrics two thresholds are set. If at any time the *delete ratio* exceeds 60%, the state machine determines that the programmer has encountered difficulty, and a query is generated, returning a set of Stack Overflow results to the developer, while the machine transitions to the *Query* state. The intuition behind this particular threshold being that if the developer is spending the majority of their time deleting text within the IDE instead of inserting it, they have encountered some form of difficulty and are need of a suggestion. In addition to the insert/delete ratio proposed by Carter and Dewan [49],

a second novel threshold introduced by this work is utilized, based on the *edit ratio*. If this ratio falls below 60%, the machine will also determine the programmer has encountered difficulty, suggest results, and transition to the query state. The rationale behind this second threshold is that if the user is spending the majority of their time moving the cursor without editing text, they have likely encountered a difficulty.

As with the window size of 25 before, the threshold of 60% was chosen heuristically. Future research may examine the effects of different threshold values and the possibility of personalizing the value of the threshold to the individual developer, an approach utilized by [50].

While in the *Query* state, the event queue is cleared and all subsequent editor events are ignored until 30 seconds has elapsed, at which point the machine transitions back to the *Collect* state. The clearing of the queue ensures that each difficulty determination is made from a clean slate, preventing events utilized in triggering past queries from influencing future difficulty detection decisions. Importantly, while in this state, no new recommendations are automatically generated. This was done to allow the developer time to process any automatically made recommendations and avoid them becoming overwhelmed.

Finally, if the user has been inactive for at least 15 minutes, i.e. no editor events are being generated, the machine transitions to a *Pause* state, where it remains until an editor event occurs. Currently, this state is utilized solely for logging purposes, however, future work may examine incorporating transitioning to and from this state into the difficulty determination decision itself.

This state machine forms the basis of the behavior-driven aspect of StackInTheFlow. Utilizing it, StackInTheFlow is able to passively listen to developer activity within the IDE and make a determination that they have encountered difficulty without their direct input. Importantly, due to the pause state having no bearing on the difficulty determination decision process, and its reliance of ratio-based metrics on a

time-independent window, the state machine is immune to many of the issues faced by other time-based approaches. Mainly that if the developer becomes preoccupied with another task, the system does not incorrectly determine that they are not making progress and thus encountering difficulty. Such an approach ensures that the system will only make such a determination if the developer is actively exhibiting behaviors that have been deemed indicators of difficulty.

Once the Difficulty Detection module has determined that the developer has encountered difficulty, either through the presence of an error message or via the state machine, the Query Generation module is triggered.

The Difficulty Detection module addresses the first issue faced by BDRSSE, that of determining if the developer is in need of a recommendation. The second module of StackInTheFlow, the Query Generation module, detailed in the following section, addresses the second challenge, that of forming a recommendation without direct input from the developer, once it has been determined that they are in need of a recommendation.

3.4.2 Query Generation

The second component of StackInTheFlow is that of the Query Generation module. This module is designed to address the second issue faced by a BDRSSE, that of formulating suitable queries to retrieve recommendations, it is responsible for enabling the functionality of several use-cases summarized below:

Difficulty Query - In the case where the Difficulty Detection module has determined that the developer has encountered some form of difficulty, the Query Generation Module must be able to formulate a query based on the context extracted from within the IDE.

Error Query - In the case where the developer encounters a compile or run time error message, the Query Generation module must be able to formulate a query

based on the extracted error message. While it is common for parts of compiler error messages and runtime stack traces to be included in Stack Overflow questions, error messages are not standardized for searching related documentation, therefore StackInTheFlow must employ a mechanism for extracting query terms from these messages.

Automatic Query - In the case where the developer explicitly invokes the query formulation process, the Query Generation module must be able to formulate a query based on the context extracted from within the IDE in a manner similar to that of the Difficulty Query use-case. However, unlike the previous use-case, the module may also extract additional context from the developer, such as in the case that they have highlighted a snippet of code to formulate the query.

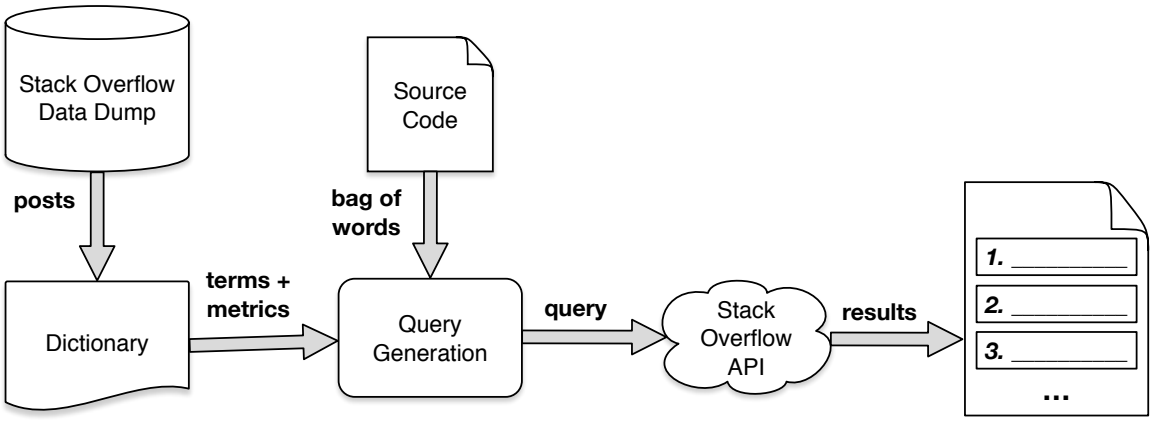
Regardless of the use-case, when the automatic query generation functionality is invoked, the Query Generation module should generate and issue a user interpretable query based on the context of the current development task. That the query is interpretable is an important quality, as it allows the developer to further refine the search query in the case that the initial generated query was not sufficient. The underlying model by which the Query Generation module does this is the same for each use-case and is described in the following section.

3.4.2.1 Query Generation Model

The Query Generation Model seeks to identify query terms which are the most *salient*, that is terms which contain the most information, with the intuition that such terms form the most suitable query terms with the highest probability of returning relevant documents. However, determining the saliency of a term is a non-trival task. To do this, the model constructs a dictionary of terms pre-mined from the user posts contained within the periodic Stack Overflow Data Dump. This dictionary contains a set of query pre-retrieval metrics [54], which enable the selection of the terms that

are most likely to retrieve a reasonable selection of documents from Stack Overflow. Since the dictionary is computed offline, query generation is lightweight and fast. An overview of the Query Generation Model is given in Figure 4.

Figure 4: StackInTheFlow Query Generation Model



The first element of the Query Generation Model is the *Stack Overflow Data Dump*[2]. This periodic data dump includes anonymized Posts, Users, Votes, Comments, PostHistory and PostLinks from a variety of sites within the Stack Exchange network. For the purposes of the model, only Posts from the Stack Overflow site are considered. Posts are provided in the form of an XML file containing several attributes, such as the post title, body, tags, or number of likes. The data dump utilized to form the most recent version of StackInTheFlow contained a total of $33,566,855$ Stack Overflow posts.

The second element of the Query Generation Model is a *Dictionary* containing potential query terms and their associated query pre-retrieval metrics. This dictionary was constructed via a custom-written XML parser implemented in Java. For each post within the data dump, the post body is extracted and converted into a bag-of-words to form a set of candidate terms. This is done by splitting the body of the post on all word boundary characters (spaces, periods, commas, etc) and then

[2]Stack Overflow Data Dump https://archive.org/details/stackexchange

converting the terms to lowercase. For a full specification of characters treated as word boundaries, see the Java Regular Expression Documentation[3]. Terms which do not begin with an alphabetic character or are less than two characters in length are discarded from consideration. From this filtered set of candidate terms the following metrics are calculated and updated as additional posts are processed:

Collection Term Frequency - The number of times a term appears within the bodies of all posts within the data dump.

Term Document Frequency - The number of posts in which a term appears.

Term Count - The total number of unique terms within the data dump.

Document Count - The total number of posts contained within the data dump.

Once all posts have been processed, for every term t an *Inverse Document Frequency* (*idf*) and *Inverse Collection Term Frequency* (*ictf*) are calculated via the following equations:

$$idf(t) = \log\left(\frac{N}{N_t}\right) \tag{3.3}$$

$$ictf(t) = \log\left(\frac{|D|}{tf(t,D)}\right) \tag{3.4}$$

where N is the document count, N_t is the term document frequency, $|D|$ is the term count, and $tf(t, D)$ is the collection term frequency.

Once all of the previous metrics have been calculated a dictionary is constructed such that for every candidate query term extracted from the data dump, the collection term frequency, document frequency, inverse collection term frequency, and inverse document frequency are recorded. This dictionary also records the overall term count

[3]Java Regex Docs https://docs.oracle.com/javase/tutorial/essential/regex/

and document count. This dictionary is then converted to an XML format and shipped with the tool itself. The current dictionary shipped with StackInTheFlow contains a total of $21,893,191$ candidate terms and is a total of 36.6MB in size, a small fraction of the approximately 40GB data dump.

The third element of the Query Generation Model is the *Query Generation Algorithm*, which takes as input a desired query length, set of candidate query terms, and the dictionary constructed within the previous element. When a query is to be generated from a source, such as an error message or source code, that source text is cleaned and converted into a bag-of-words in a process detailed in Section 3.4.2.2. Once extracted, this bag-of-words is treated as a set of candidate query terms and is fed into the algorithm as input. Each candidate term is scored on a variety of pre-retrieval metrics [54] detailed below:

Specificity - The specificity of a query is a measure of the distribution of the terms within the corpus. For queries composed of generic terms which occur frequently within the corpus, for example "the" and "there", the quality is deemed lower than that of queries containing specific terms such as "apache" or "spark". To assess the specificity of the candidate terms the Inverse Document Frequency and the Inverse Collection Term Frequency for each term within the dictionary is utilized.

Similarity - The similarity of a query is a measure of the similarity between its terms and the corpus, with the intuition that queries judged similar to the corpus are easier to answer and thus of higher quality. For each candidate term, the *Collection Query Similarity* (SCQ) metric [55] is utilized to ascertain a similarity score. The Collection Query Similarity of a term is calculated via the following equation:

$$SCQ(t) = (1 + \log(tf(t, D))) \cdot idf(t) \tag{3.5}$$

This metric measures the vector-space based query similarity to the collection.

It does this by considering the collection as one large document, composed of a concatenation of all the documents.

Coherence - The *coherence* of a query is a measure of the inter-similarity of documents containing the query terms. In order to be practical, heavy analysis must be performed during indexing time. For each term in the index vocabulary, a coherence score, $CS(t)$ is calculated. This score represents the average pairwise similarity between all pairs of documents in D_t, the set of documents which contain t:

$$CS(t) = \frac{\sum_{(d_i,d_j)\in D_t} sim(d_i, d_j)}{|D_t|(|D_t| - 1)} \tag{3.6}$$

where $sim(d_i, d_j)$ is the cosine similarity between the vector-space representations of the two documents. Several metrics have been proposed to examine this dimension, however most are computationally expensive to compute as they require a pointwise similarity matrix for all documents contained within the corpus. StackInTheFlow instead employs a less expensive metric proposed by Zhao et al. [55], $VAR(t)$, which measures the variance of the query term weights over the documents which contain them. To compute the weight of each query term a *tf-idf* based approach is utilized given by the following formula [56]:

$$w(t, d) = \frac{\log(1 + tf(t, d)) \cdot idf(t)}{|d|} \tag{3.7}$$

The argument behind $VAR(t)$ follows that if the variance is low, it will be more difficult to differentiate between relevant and irrelevant documents, thus the overall query quality is low.

For each candidate source code term contained within the dictionary, we calculate the above metrics treating each term individually as a query, and then linearly sum the score for each dimension to achieve an overall query term score ($QS(t)$). Thus, $QS(t)$

is given by the following equation:

$$QS(t) = idf(t) + ictf(t) + SCQ(t) + VAR(t) \tag{3.8}$$

In the current implementation of StackInTheFlow, all query pre-retrieval metrics are weighted equally, however, future research may examine tailoring the weights of each metric to suit the individual user. Once all candidate terms have been scored, the top n scoring terms are then selected to form the candidate query. By default, StackInTheFlow utilizes a value of 4 for n.

3.4.2.2 Query Generation Model Implementation

Though the underlying model by which the Query Generation module formulates a query is the same, the method by which candidate query terms are extracted from the IDE context varies between each use-case. The following section describe this process for each use-case.

Difficulty Query - When invoked via the difficulty query use-case, the Query Generation Model is utilized to formulate a query based on the context extracted from within the IDE. This is done by focusing on two pieces of information extracted from the currently open source code file within the editor. The first being any `import` statements present at the top of the source code file, the second being the line of code on which the cursor currently resides. To extract candidate query terms from Java-style import statements, each package level of the statement is treated as a separate term. In addition, all terms are converted to lower case to ensure compatibility with the term dictionary. For example, the import statement `import org.apache.spark.sql.SparkSession` would produce the following set of query terms: *org*, *apache*, *spark*, *sql*, *sparksession*. To extract candidate query terms from a line of source code a similar process is utilized. A line of source code is split across word boundary characters and the resultant terms are converted to lower case. For example, the following line of code

`System.out.println("Hello World")` would produce the following set of query terms: *system*, *out*, *println*, *hello*, *world*. These two sets of candidate terms are then combined and sent to the Query Generation Model to form a candidate query.

Error Query - When invoked via the error query use-case, the Query Generation Model is utilized to formulate a query based on information extracted from the *stacktrace* of the error. For example consider the following stacktrace:

```
Exception in thread "main" java.util.NoSuchElementException
    at java.util.Scanner.throwFor(Scanner.java:862)
    at java.util.Scanner.next(Scanner.java:1371)
    at Main.main(Main.java:23)
```

This stracktrace is cased by a runtime error due to the user attempting to read in a blank line with a Scanner. To formulate a query from this stacktrace two separate approaches are utilized, one which relies on the Query Generation Model and one which bypasses it. The first approach bypasses the model an instead utilizes the first line of the stacktrace directly as a query, in this example “Exception in thread "main" java.util.NoSuchElementException”. This is done due to the prevalence of the first line of an error message in many Stack Overflow posts. If the first approach fails to retrieve any results, a second approach similar to the difficulty query formulation process is utilized. The stacktrace is converted into a bag-of-words by splitting the entirety of its text on word boundary characters and converting the resultant terms to lower case. In addition, regular expressions are utilized to extract the exception classes and language version specifically. These terms are then fed into the Query Generation Model to form a candidate query.

Automatic Query - When invoked via the automatic query use-case, the Query Generation Model is utilized to formulate a query based on the context extracted from within the IDE at the behest of the user. This means that in addition to the information available previously, additional cues in the form of selected regions of text can be provided by the user to assist in the query generation process. When invoked, if the

user has not selected any region of text, a process identical to that used in the difficulty query case is employed. Candidate query terms are extracted from `import` statements and the line of code on which the cursor resided in the editor as before. However, if the user has chosen to select a region of text prior to invoking query generation, this process is discarded in favor on one utilizing the selected text. Instead of extracting candidate query terms from `import` statements or the current line of code as before, the entirety of the selected text is instead utilized to form candidate query terms. Mirroring the other use-case approaches, this selected text is converted into a bag-of-words by splitting it on word boundary characters and converting each term to lowercase. This set of candidate terms is then fed into the Query Generation Model to form a candidate query.

Regardless of the use-case, when a candidate query is utilized which has been generated by the Query Generation Model, if the query fails to retrieve any results a back-off technique is employed by removing the lowest scoring term from the query and re-initiating a document search. This process is repeated until a query of one term is reached, at which point results are guaranteed based on the reliance of terms within the dictionary extracted from the Stack Overflow Data Dump. For example, given an initial query of "spark apache java" generated by the Query Generation Model, a document search would be initiated. If no results were found a query of "spark apache" would be utilized. If still no results were found a query of "spark" would be utilized. At this point results would be guaranteed due to the presence of the term *spark* within the model's dictionary, as there *must* be a least one post which contains the term.

All results deemed relevant to a particular query are retrieved via the Stack Overflow API, described in the following section.

3.4.3 Stack Overflow API

The third component of StackInTheFlow is the Stack Overflow API. This competent is responsible for querying Stack Overflow with queries generated by the Query

Generation module, or those made directly by the user. Thus, this component enables the manual query use-case. The Stack Overflow API is part of the larger Stack Exchange API[4] which is a RESTful API that enables the querying of sites within the Stack Exchange Network for various pieces information such as user data, or information pertaining to the questions, answers, and comments within the site. It also provides functionality for posting new topics to the sites within the network. Importantly, the API provides a `search/advanced` endpoint which enables the retrieval of relevant posts given a search query. StackInTheFlow utilizes this endpoint to query Stack Overflow to retrieve documents relevant to a particular query. When querying for relevant documents, the endpoint provides four possible schemes by which results can be ranked summarized below:

Relevance - ranks retrieved documents based solely on their computed relevancy to the query. Relevancy is determined by an unspecified black-box algorithm provided by the API.

Creation - ranks retrieved documents based on their creation date, newer documents will appear higher in the ranking than older documents.

Votes - ranks retrieved documents by the number of votes they have received on the Stack Overflow website, documents with more votes will appear higher in the ranking than those with fewer votes.

Activity - ranks retrieved documents by their last activity date. Activities include events such as a user posting a comment or answer to an initial question. Documents which have a newer last activity date will appear higher in the ranking than documents which have become inactive.

All four ranking schemes are exposed to the user, which may select which scheme they

[4]Stack Exchange API https://api.stackexchange.com

would prefer StackInTheFlow to utilize from the user interface. When a query is sent to the API, either directly by the user or via the Query Generation module, based on the selected scheme a ranked listing of relevant Stack Overflow articles will be retrieved. In the case that there are no articles relevant the query, an empty listing will instead be returned. In the case that there are relevant articles, the retrieved ranked listing with then be rendered in the user interface for the developer to view. If the user has selected the *relevance* ranking scheme, the order in which the results are presented will be re-ranked in a procedure described in Section 3.4.4, otherwise they will be presented in the order given by the Stack Overflow API.

3.4.4 Result Personalization

The fourth component of StackInTheFlow is that of the Result Personalization module. If the user has selected the *relevance* ranking scheme, StackInTheFlow personalizes the results of all queries by re-ranking the results based on past user activity. It does this by utilizing a novel metric, *Click Frequency-Inverse Document Frequency* (*cf-idf*), which aims to predict the affinity of a developer towards a retrieved Stack Overflow post by analyzing the tags associated with previously clicked results by that developer. *Click Frequency-Inverse Document Frequency* is composed of two constituent metrics. The first, *Click Frequency* (*cf*), is computed for a given tag t on a given result selection history H. Using raw frequency $f_{t,H}$, defined as the number of times a Stack Overflow post tagged with t was clicked on by the user, as recorded in H, the corresponding *Click Frequency* is given by the following equation.

$$\mathrm{cf}(t, H) = \begin{cases} 1 + \log(f_{t,H}) & f_{t,H} > 0 \\ 0 & f_{t,H} = 0 \end{cases} \tag{3.9}$$

The second metric is the *Inverse Document Frequency* (*idf*) of a tag, is computed from the Stack Overflow Data Dump by the following equation:

$$idf(t) = \log\left(\frac{N}{N_t}\right) \tag{3.10}$$

where t is a given tag and N_t is the number of articles associated with that given tag within the data dump. This metric has the effect of discounting tags that are prevalent in the corpus across multiple documents. From these two metrics *Click Frequency-Inverse Document Frequency* (*cf-idf*) is calculated via the following equation:

$$cfidf(t, H) = cf(t, H) \cdot idf(t) \tag{3.11}$$

Such a metric falls into the *Content-Based Filtering* approach to recommendation system ranking schemes. In this context, the set of all available tags represent the topics available to recommend. Clicks on articles associated with a particular tag represent a higher rating of that topic by the user. In this way, content-based filtering can be applied to score a result tagged with topics the user has rated more highly with a more favorable score than one which does not share previously highly rated tags. The procedure by which this is done is described below.

Given an initial set of of ranked results retrieved from Stack Overflow, for each result a raw score (S) is computed by taking the sum of the *cf-idf* of each tag associated with the result and dividing by the total number of tags associated with the result. This score is computed via the following procedure:

1. Let T be the set of all available tags in a collection of documents D.

2. Define a **User Vector**, $\vec{U}$ as a vector containing the cf-idf of every tag $t \in T$ given H.

3. Define a **Result Vector**, $\vec{r}$ as a binary vector for a given result r where $\forall t \in T$, $\vec{r}_t = \{1 \text{ if } r \text{ is tagged with } t : 0 \text{ if } r \text{ is not tagged with } t\}$.

4. Let g be the set of a tags associated with a result r.

5. The raw score (S) of a given result r is given by the following equation:

$$S(r) = \frac{\vec{U} \cdot \vec{r}^{\top}}{|g|} \tag{3.12}$$

To determine the final position of a result the following procedure is utilized:

1. Let I be the initial ranking of results.

2. Let W be a weighted ranking of results in descending order based solely on the raw score for each result.

3. The adjusted score P of a given result r is then calculated via the following equation:

$$P(r, I, W) = \frac{r_i + r_w}{2} \tag{3.13}$$

   where r_i is the initial ranking of result $r \in I$ and r_w is the weighted ranking of result $r \in W$.

4. The final position each result is determined by re-ordering the set of results by descending adjusted score.

Utilizing this approach, both the ranking determined via content-based filtering and the original ranking provided by the Stack Overflow API are taken into consideration when determining the final ranking of a result. An example of this approach is demonstrated in Section 3.4.4.1.

3.4.4.1 Personalization Ranking Example

In this example there are 5 tags within our corpus *A*-*E* ($|T| = 5$). A sample result section history H is also provided. The *idf* of each tag is also given as an estimation from the corpus. These base metrics are shown in Table 2. A calculation of $\vec{U}$ is also given.

$$\vec{U} = (6.53, 4.93, 7.59, 5.07, 7.85)$$

Table 2.: Basic Personalization Metrics

Tag (t)	H	cf(t,h)	idf(t)	cfidf(t,H)
A	2	1.30	5.02	6.53
B	1	1.00	4.93	4.93
C	3	1.48	5.14	7.59
D	1	1.00	5.07	5.07
E	4	1.60	4.90	7.85

Five results have also been returned from the Stack Overflow API R_1-R_5 ($|I| = 5$) each with a subset of tags associated with them. The computation of $\vec{r}$ and $S(r)$ for each result is shown in Table 3. From the the calculation of $S(r)$ one can obtain W

Table 3.: Calculation of Personalization Raw Score

Result (r)	g	$\vec{r}$	$S(r)$
R_1	*A,E*	$(1,0,0,0,1)$	7.19
R_2	*C,D*	$(0,0,1,1,0)$	6.33
R_3	*B,E*	$(0,1,0,0,1)$	6.39
R_4	*B,D*	$(0,1,0,1,0)$	5.00
R_5	*A*	$(1,0,0,0,0)$	6.53

a weighted ranking of the results. Once a weighted ranking has been determined, an adjusted score can be computed. Finally, from the calculation of the adjusted score a final ranking (F) is obtained, given by Table 4. This leads to a final result ranking of R_1, R_2, R_3, R_5, R_4 based on the user's preferences.

As illustrated by this example, the final ranking does not differ drastically from the original, re-ordering only two terms. If the weighted ranking had been utilized alone to determine the final ordering, the re-ranking of results would have been more

Table 4.: Calculation of Personalization Adjusted Score & Final Ranking

Result (r)	I	W	$P(r, I, W)$	F
R_1	1	1	1.00	1
R_2	2	4	3.00	2
R_3	3	3	3.00	3
R_4	4	5	4.50	5
R_5	5	2	3.50	4

drastic. The inclusion of the original ranking in the the computation of the final ranking ensures that the ranking does not significantly differ from the original unless the user has exhibited a very strong preference for the topics associated with a result which had been initially lower ranked. Once more, any additional re-ranking performed is most likely to be in the form of small distance jumps of results within the ranking, with large adjustments to the original order being rare.

CHAPTER 4

RESULTS AND DISCUSSION

To evaluate the effectiveness of StackInTheFlow, two different approaches were utilized.

4.1 Evaluation Approach I: Collection of Anonymous Logs

In the first approach, anonymous logs of user interactions with the tool have been collected. In the period between August and November of 2017, StackInTheFlow, with all the features described in this thesis, has been publicly available for download from the JetBrains tool repository. Following advertising of the tool on various channels, it has been downloaded 148 times, with logs reflecting tool use captured from 77 unique users. Logs from StackInTheFlow developers have been marked and removed.

The goal of this evaluation is to estimate the effectiveness of each query type. To perform this evaluation, reliance is made on the tool feature usage and click-through rates found in the logs. The assumption is that clicking on a query's result to open it in the browser or expand its content for reading within StackInTheFlow is an indication of its effectiveness.

4.2 Evaluation Approach II: Observing Developers with StackInTheFlow

In the second approach, a small study consisting of 5 undergraduate computer science students who had all successfully completed an Android development course, recruited from friends and colleagues, was performed in which participants were tasked with completing an Android development task with access to StackInTheFlow, during which recordings of their interactions with the tool were made for observation.

The goal of this evaluation is to further estimate the effectiveness of each query

type and to asses the overall usefulness of StackInTheFlow. This was done through the combination of annotating the recordings made of the participants interactions with the tool and asking the participants to complete a post-study questionnaire designed to asses the overall usefulness of StackInTheFlow. While determining a scheme of recording annotations, emphasis was placed on annotating indicators that the results returned by the tool were useful to completing the assigned task.

To begin the study, participants were first asked to complete a pre-study questionnaire consisting of the following questions designed to asses their programming experience:

1. *How many years of programming experience do you have?*
2. *How many years of Java programming experience do you have?*
3. *How many years of Android programming experience do you have?*
4. *How strong are your Java development skills?*
5. *How strong are your Android development skills?*

Participants were also asked to rate on a 5-point Likert scale from 1 (never) to 5 (very often) how often they utilized Stack Overflow when encountering the following development situations:

1. *Getting "stuck" in development*
2. *Encountering a compiletime error*
3. *Encountering a runtime exception*
4. *As a reference for an API and its use*

Finally, participants were asked the following two questions regarding their information searching behaviors:

1. *Are there any other situations (besides those listed above) in which you utilize Stack Overflow?*

2. *What other sources of information on the Web do you utilize when solving development issues besides Stack Overflow?*

Table 5.: Participant Reported Years of Programming Experience

	P_1	P_2	P_3	P_4	P_5	**average**
Android	1	1	1	0.5	3.5	**1.4**
Java	2	7.5	3	4	4	**4.1**
Overall	2	8.5	3	5	4.5	**4.6**

Table 5 shows the reported years of programming experience for each of the five participants. Participants reported an average of 4.6 years of programming experience, with an average of 4.1 years of Java experience and 1.4 years of Android experience. 3 of 5 participants reported 1 year of Android experience, with one reporting 0.5 years and one reporting 3.5 years. Participants also on average rated their Java development skills as 4 out of 5 and their Android development skills as 2.8 out of 5.

In regards the the second set of questions, participants reported on average a score of 4 out of 5 to getting "stuck", 4.4 out of 5 to compiletime errors, 4.4 out of 5 to runtime exceptions, and 2.6 out of 5 to an API reference.

Finally, most participants did not offer additional situations in which they utilized Stack Overflow and cited Google as the most often utilized additional source of information.

Once the pre-study questionnaire was completed, participants were asked to progress as far as possible in an Android development task taken from the commit history of the open-source app NextCloud Notes[1]. NextCloud Notes is a simple note taking app in

[1]NextCloud Notes https://github.com/stefan-niedermann/nextcloud-notes

which users may create and edit textual notes and then export them to various online services such as Google Drive. Participants of the study were tasked with adding a splashscreen during the loading process of the app via the following prompt:

> *Currently, there is no splash screen. Not only are we missing out on a chance for branding, but there is a slight lag when the user opens the app due to server connection latency.*
>
> *Help improve the user experience by creating a splash screen that displays when the user starts the app. It should help hide the latency between the app and the server.*

To complete this task, participants were alloted 1 hour. Before beginning, participants were given a brief demo of StackInTheFlow so that they were informed of its various features. However, participants were not required to utilize StackInTheFlow to complete the task and were free to use any available online resources (Google, YouTube, Blogs, etc.) with the exception of the online source code of NextCloud Notes (which contained a solution to the task). Throughout the duration of the task, a recording of their computer screens was made. These recordings were then collected to be annotated.

Annotations were broken into three categories: *session*, *query*, and *interact*. A full breakdown of each of the annotations available in each category are given by Table 6.

The *session* category refers to the period of time between the participant beginning a search for information, denoted by the `start` tag, and when they have either given up, denoted by the `end_fail` tag, or when the have successfully utilized knowledge from their search in their development task denoted by the remaining tags. If the user utilized an external resource, such as Google, to extract this information it is denoted by the `end_external` tag, otherwise the ending tag appropriate to the type of query which generated this information is utilized.

The *query* category refers to when a query is issued, either through the tool, in which case the type of query is denoted, or through some external resource, in which

Table 6.: Recording Annotations

session	*query*	*interact*
`start`		
`end_fail`		
`end_external`	`external`	`external`
`end_manual`	`manual`	`manual`
`end_auto`	`auto`	`auto`
`end_error`	`error`	`error`
`end_diff`	`diff`	`diff`

case the `external` tag is utilized.

Finally, the *interact* category refers to when the user interacts with the result of a query, such as clicking on a result. In such a case, the type of query which generated the result is denoted.

Recordings were annotated independently by both the author and his advisor. After all recordings had been annotated, the sets of annotations for each recording were compared and any divergent annotations discussed. This happened within three of five recordings. The divergent annotations were able to be easily resolved after a second examination of the point in the recordings in which they occurred.

At the conclusion of the study, participates were asked to complete a post-study questionnaire. In the first part of the questionnaire, designed to detect problems with the experimental design, participants were asked to express their level of agreement on a Likert scale from 1 (absolutely no) to 5 (absolutely yes) to the following claims:

1. *The activity to perform was clear overall.*

2. *The individual tasks to perform were clear.*

3. *There was enough time alloted to perform each task.*

4. *The tasks were easy to complete.*

In the second part of the questionnaire, designed to collect qualitative information regarding the usefulness of StackInTheFlow, participants asked to answer the following questions:

1. *How often did you use StackInTheFlow?* Possible answers used a 5-point Likert scale from 1 (never), 2 (in 25% of searches), 3 (in 50% of searches), 4 (in 75% of searches), 5 (always).

2. *StackInTheFlow had the capability to automatically generate queries when it determined you were stuck, or via a right-click action. In such cases if they occurred, how helpful were the queries generated?* Possible answers used a 5-point Likert scale from 1 (very unhelpful) to 5 (very helpful).

3. *If applicable, in the case that StackInTheFlow determined you were stuck and thus automatically generated a query, how appropriate was the timing of this suggestion?* Possible answers used a 5-point Likert scale from 1 (very inappropriate) to 5 (very appropriate).

4. *How would you improve StackInTheFlow?*

4.3 Research Questions

The objective of both approaches is to asses the effectiveness of each query type in assisting the developer to complete their tasks. For a complete description of each query type, see Chapter 3. Thus the research questions they attempt to address are as follows:

RQ_1: *How effective are manual queries in assisting the developer?*

RQ_2: *How effective are auto queries in assisting the developer?*

RQ_3: *How effective are error queries in assisting the developer?*

RQ$_4$: *How effective are difficulty queries in assisting the developer?*

4.4 Results

The results of each evaluation method are presented below.

4.4.1 Evaluation Method I Results

Figure 5: Logged Query Types

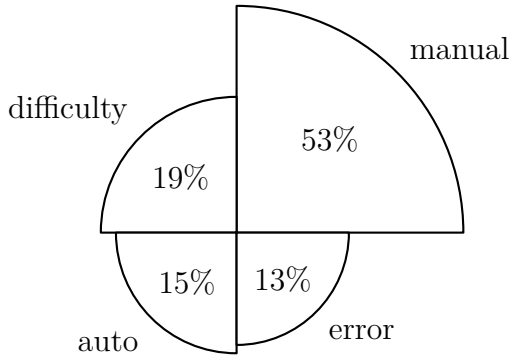
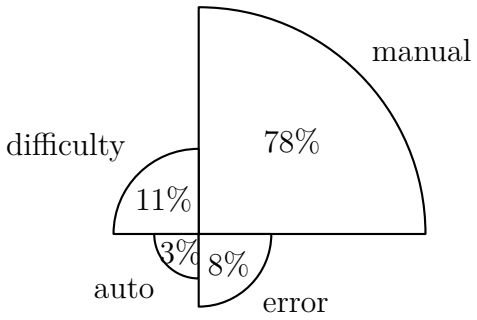


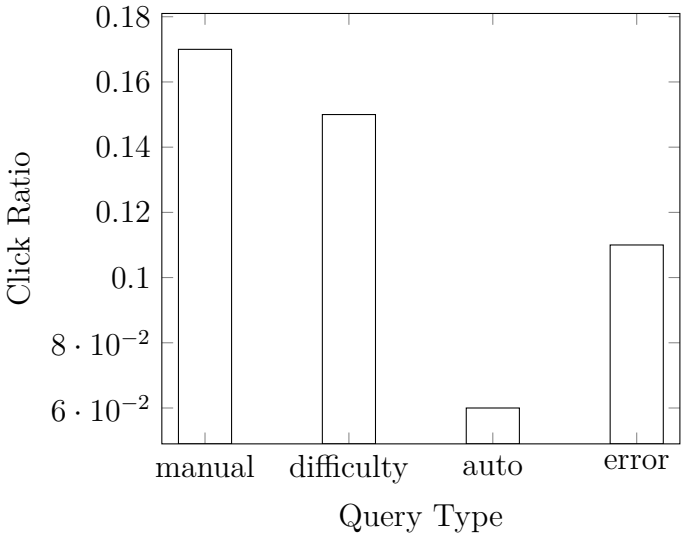
Figure 5 illustrates the percentage of each query type present in the log. Out of the 794 queries logged, 53% came from manual user input, 19% from difficulty detection, 15% from auto queries, and 13% resulted from error message queries. The manual query is utilized as the baseline query type against which the other types are compared.

Figure 6: User Interaction per Query Type



The logs contain 214 instances of user-result interactions, defined as the user viewing the result of a query in the browser, or expanding or contracting a result. Out

Figure 7: Ratio of Clicks to Total Number of Queries per Query Type



of these, users opened the browser 115/214 (54%) times, and either expanded or contracted articles within the tool 99/214 (46%) times. Figure 6 illustrates the proportion of user-result interactions per query type. In correlating the StackInTheFlow user-result interactions with a query type, it was observed that 166/214 (78%) are on manual queries, 7/214 (3%) are on automatically generated queries, 24/114 (11%) are on difficulty queries, while error queries account for 17/214 (8%) of user interactions.

Finally, it was explored how often users interact with retrieved Stack Overflow posts based on query generation type. Figure 7 illustrates the ratio between the number of times a user clicked on a result and the total number of queries issued for that query type. Out of 426 manual queries, 99 (17%) had at least one click. From 117 automatically generated queries, 7 (6%) had at least one click. Out of 149 difficulty queries, 23 (15%) had at least one click. Out of 102 error queries, 11 (11%) had at least one click.

In summary, these results indicate that automatic queries rarely received user interaction. Both error (11%) and difficulty (15%) queries resulted in at least a single click with a frequency on par with manual queries (17%), which indicates that they

were reasonably effective.

4.4.2 Evaluation Method II Results

Of the five participants, three (P_1, P_3, P_5) were able to successfully complete the task of adding a Splash Screen to NextCloud Notes at startup. The time taken by each participant to work on the task is given by Table 7.

Table 7.: Time in Minutes Utilized per Study Participant

P_1	P_2	P_3	P_4	P_5
53:35	60:00	43:56	60:00	38:59

It can be observed that of the participants that successfully completed the task, all required more than half the alloted time to do so. Thus, the task was non-trivial and required sufficient effort on behalf the participants to complete. This is important as it allows the task to simulate a more "real-world" development scenario in which the developer is unfamiliar with the task at hand and must rely on external resources to complete it.

The remaining results of this evaluation method are divided into those extracted from the annotations of the recordings of the screen of each participant while completing the Android development task and those collected from the post-study questionnaire.

4.4.2.1 Recording Annotation Results

The frequency counts of each *query* tag for each participant as well as their average across all participants is given by Table 8. It can be observed that participants issued mostly external queries to outside resources such as Google while completing the task, followed by manual queries through StackInTheFlow. Difficulty queries were issued with varying frequency by the tool. Interestingly, more difficulty queries were issued for the participants who were unable to successfully complete the task. Only

Table 8.: Query Annotation Frequency per Study Participant

query	P_1	P_2	P_3	P_4	P_5	**average**
`external`	6	14	5	10	12	**9.4**
`manual`	8	10	3	4	1	**5.2**
`auto`	0	0	1	2	0	**0.6**
`error`	0	0	0	0	0	**0**
`diff`	1	4	1	5	1	**2.4**
sum	15	28	10	21	14	**22**

two participants (P_3, P_4) utilized the auto query feature leading to a relativity low average frequency. Unfortunately due to a limitation in the implementation of StackInTheFlow, error queries were not able to be triggered by the tool for error messages generated by the Android Emulator and thus though a small number of errors did occur during the participants development of a solution to the task, no error queries were issued.

Also of interest is the total number of queries issued by each participant. In total participants issued an average of 22 queries. However, participants that were unable to complete the task issued on average 11.5 more queries in total than participants who were able to successfully complete the task.

In addition to the frequency of each query type, the frequency of participant interactions with the results, such as clicking on a result to view more information or expanding the result in the tool, of each query type may also be observed. Table 9 gives the frequency of each *interact* tag per participant as well as an average across all participants.

As to be expected, queries issued at higher frequencies produced a higher quantity of interactions. It can also be seen that participants interacted with almost all external and manual queries they issued. However, they did not interact with any auto queries

Table 9.: Interact Annotation Frequency per Study Participant

interact	P_1	P_2	P_3	P_4	P_5	**average**
`external`	7	14	3	18	11	**10.6**
`manual`	6	6	3	3	0	**3.6**
`auto`	0	0	0	0	0	**0**
`error`	0	0	0	0	0	**0**
`diff`	0	0	0	0	1	**0.2**
sum	13	20	9	21	12	**15**

and only one user (P_5) interacted with a difficulty query.

Finally, the frequency of success of each session may be observed, given by Table 10. Interestingly, a large majority of sessions ended in failure, with on average 76% of sessions ending in failure.

Table 10.: Session Annotation Frequency per Study Participant

session	P_1	P_2	P_3	P_4	P_5	**average**
`start`	15	28	10	21	14	**17.6**
`end_fail`	10	24	6	19	11	**14**
`end_external`	4	1	2	1	3	**2.2**
`end_manual`	1	3	2	1	0	**1.4**
`end_auto`	0	0	0	0	0	**0**
`end_error`	0	0	0	0	0	**0**
`end_diff`	0	0	0	0	0	**0**

Surprisingly, when the average success rate of manual versus external queries is examined, given by Table 11, it can be observed that manual queries from the tool achieved an average success rate on par with that of external queries, despited being

constrained to a search corpus of only Stack Overflow. The very low manual success rate of participant 5 is due to them issuing only one manual query, which failed to return any useful information. If this outlier is omitted, the average success rate of manual queries increases to 33.5%, slightly above that of external queries.

Table 11.: External vs Manual Query Success Rate

query	P_1	P_2	P_3	P_4	P_5	**average**
external	66.7%	7.1%	40%	10%	25 %	**29.8%**
manual	12.5%	30%	66.7%	25%	0%	**26.8%**

This success does not extend to the other query types however, which either had a success rate of 0%, or one which is undefined. It is however difficult to draw a conclusion from this result though, due to the small number of queries issued of these types.

In summary, the results of this evaluation method indicate that manual queries are reasonably effective at assisting the developer, however results are inconclusive for auto and difficulty queries due to a small sample size, and unavailable for error queries due to a limitation in the implementation of StackInTheFlow.

4.4.2.2 Post-Study Questionnaire Results

The responses to the Post-Study questionnaire provided interesting insights into the participants' impressions of the task and of StackInTheFlow. In regards to the prompt inquiring if the task to perform was clear overall. The three participants (P_1, P_3, P_5) who were able to successfully complete the task responded 5 (absolutely yes), while the two who were unable to complete the task (P_2, P_4) responded 4 (mostly yes). A similar response is given to the prompt inquiring if the individual tasks necessary to complete overall objective were clear, with the three participants able to successfully complete it again responding 5, and the two unable to complete it responding with 4 and 3 (neither yes nor no). Interestingly, all participants except participant 5 responded

with a 5 when asked if there was enough time to complete the task, even those that were unable to successfully complete it. Surprisingly participant 5, who completed the task in the least amount of time responded with a 4. Finally, when asked if the assigned task was easy to complete the two participants who did not complete the task responded with a 3. The participants who successfully completed the task were split with two responding 4 and one responding 5.

In regards to the prompts assessing the overall usefulness of StackInTheFlow when asked to self-rate how often they utilized the tool to perform queries two participants (P_1, P_4) responded with 50% of the time, and the remaining participants responded with 25% of the time. Interestingly, this does not quite align with their actual usage of the tool, with P_4 overestimating their use of StackInTheFlow and P_2 underestimating their use.

Figure 8: Study Participant Rating of the Usefulness of Difficulty and Auto Queries

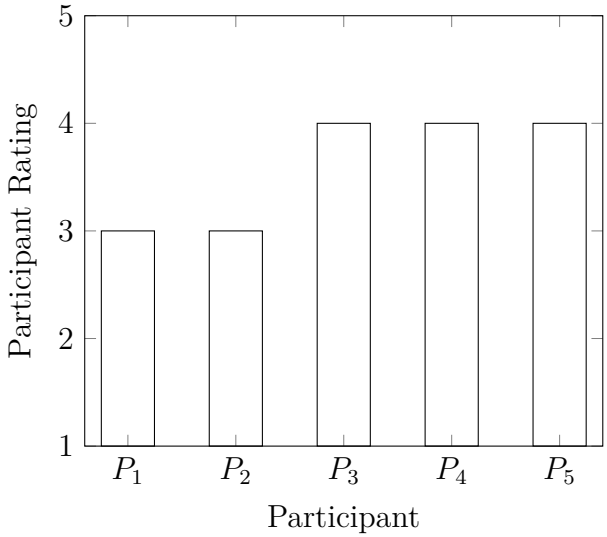


Figure 8 shows participants' reported ratings in response to the prompt inquiring as to the usefulness of difficulty and auto queries. In contrast to the annotations, which deemed that no such queries were successful, the majority of study participants rated the results returned by these query types as somewhat helpful. When asked to rate the

appropriateness of the timing of difficulty queries however, the results are more mixed.

Figure 9: Study Participant Rating of the Timing of Difficulty Queries

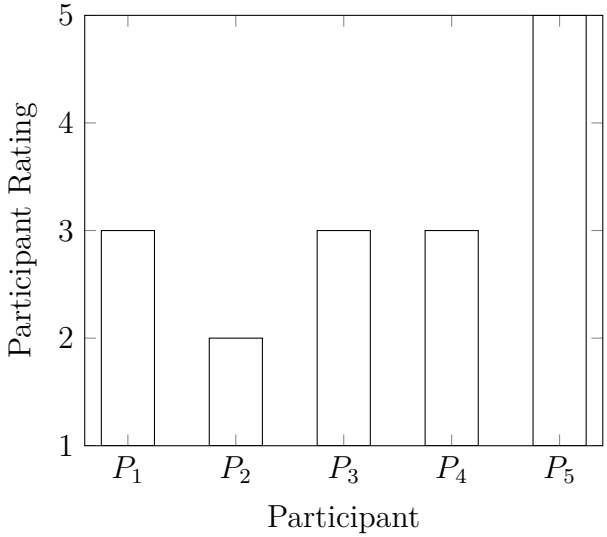


Figure 9 reports the rating of each participant. Most participants gave a rating of 3 (neutral) to the timings. However, participant 2 gave a rating of 2 (somewhat inappropriate) and participant 5 gave a rating of 5 (very appropriate). Participant 2 experienced four difficulty queries whereas participant 5 experienced only 1. The lower rating of participant 2 may be due to unawareness that a difficulty had taken place, as they reported in response to the final question that they desired the triggering of a difficulty query to be more prominently displayed in the UI. Participant 5 on the other hand, appears to think that the timing of the one difficulty query they experienced was very appropriate.

When asked how they would improve the tool, participants offer up improvements to the user interface such as (i) adding more methods of invoking the tool, such as in the tooltip help dialog or in the context menu or (ii) more prominently displaying when a difficulty query is performed. The participants also expressed a desire to improve the quality of queries and results such as (iii) improving the quality of auto query keywords and (iv) filtering out unrelated results such as those pertaining to Android

within alternative IDEs like Eclipse.

In summary, the results of this post-study questionnaire indicate that participants felt that the task they were assigned was achievable within the time alloted. Furthermore, that the queries automatically generated by StackInTheFlow were somewhat helpful, even if their timing was not always appropriate. There are also areas in which the tool can be improved such as a better user interface which allows more methods of tool interaction and more clearly displays when the tool has initiated a query on the user's behalf. The quality of automatically generated queries and the results retrieved by the tool could also be improved.

4.5 Discussion

To analyze the results of both methods of evaluation, they are first examined in relation to each of the four research questions. Afterwards, a discussion of meaningful observations from the recordings collected in the second evaluation method are given.

4.5.1 RQ_1: How Effective are Manual Queries in Assisting the Developer?

As manual queries were used as a baseline in Evaluation Method I, it is hard to ascertain their effectiveness from its results alone other than to state that manual queries exhibit the highest click ratio compared to the other methods of queries offered by StackInTheFlow. However, examining the results of Evaluation Method II, manual queries achieve a success rate on par or exceeding that of queries issued to external resources such as Google. Given the objective of StackInTheFlow is to provide a resource for developers to gain access to information while remaining in the flow of the IDE environment, the high success rate of manual queries indicates that this objective has been at least partially met. Thus, it can be concluded that manual queries are reasonably effective at assisting the developer.

4.5.2 RQ_2: How Effective are Auto Queries in Assisting the Developer?

Based on the results of Evaluation Method I, auto queries had the lowest click ratio of the four query methods. In Evaluation Method II, only two of the five participants utilized an auto query, of which none were successful. One possible explanation for the low performance of auto queries is due to their sole dependence on source code as a method of extracting context. The quality of the terms available within the source code will have a profound effect of the quality of queries produced. In the case of Evaluation Method II, often generic terms such as "ICallback" appeared in auto generated queries. Such terms relate to a wide variety of frameworks and languages beyond Android and Java. As a consequence, the quality of results suffered. It was also observed that study participants would often use an auto generated query as the basis of their own, modifying it into a manual query, sometimes utilizing it in an external search engine such as Google. Thus, though the initial query was not very fruitful, it served as the basis of a more useful one.

4.5.3 RQ_3: How Effective are Error Queries in Assisting the Developer?

Due to the limitations of Evaluation Method II, only results from Evaluation Method I may be utilized to asses the effectiveness of error queries. However given this, it can be observed that they achieved a comparable 11% click ratio to that of manual queries (17%). These results indicate that they are reasonably effective in assisting the developer.

4.5.4 RQ_4: How Effective are Difficulty Queries in Assisting the Developer?

The set of results pertinent to difficulty queries represents the largest subset of collected results. In Evaluation Method I, they achieved a surprisingly high click ratio of 15%, second only to manual queries. The results returned by difficulty queries were rated as mostly helpful by the participants of Evaluation Method II, even if their timings

could be improved. Despite this, no difficulty query was annotated as being successful. One possible explanation for this may be the relatively low amount of difficulty queries issued during the duration of the study. Another possible issue is that the participants simply did not notice when a difficulty query had been issued, as noted in their responses to the study post-questionnaire. Difficulty queries are further complicated in that they need to be issued by the difficulty detection model. This mechanism however, appears to be working as intended, as more difficulty queries were issued (and thus difficulty detected), for participants who were not able to successfully complete the task. The reason for their poor performance in the study may then lie in that difficulty queries are also susceptible to the same issues regarding query generation and quality of results as auto queries. If the code contains generic terms, as it did in the study, the results returned may be of sub-optimal quality. In spite of this, it appears that on a larger scale and across more domains than those covered in the study, difficulty queries are producing clicks at a rate on par with manual queries and thus can be considered moderately effective with the caveats previously discussed.

4.5.5 Additional Observations

During the process of annotating the recordings of participants' screens in Evaluation Method II, several interesting observations were made as to the effectiveness and limitations of StackInTheFlow. A selection of these observations are presented below.

Observation I: Stack Overflow is not appropriate for exploratory searches - It was observed that when each study participant began the task, they often started by searching some form of the query "Android Splash Screen", however, doing so did not return any relevant Stack Overflow articles. Participants then repeated this query in Google and visited blogs and tutorial sites describing a method of adding splash screens to Android applications. When Stack Overflow articles were used by the participant it

was in regards to more specific queries such as “Start new Android Activity”.

This observation illustrates a limitation of the Stack Overflow corpus. It is more suited to answering specific issues than describing the overarching technique for implementing a specific feature. Due to StackInTheFlow's sole reliance on Stack Overflow to retrieve results it also shares in this limitation. This means that for a novice developer first beginning a development task, they may be better served by a blog or tutorial site until they develop specific questions more suited to be answered by Stack Overflow.

Observation II: Google searches performed better than Stack Overflow API searches on the same content - It was also observed that on numerous occasions study participants would issue a manual query to StackInTheFlow but be unsatisfied with the returned results, they would then proceed to copy and paste the query, unmodified, into Google to retrieve higher quality results. Many times these results were also Stack Overflow articles.

This observation illustrates that the information retrieval algorithm utilized by the Stack Overflow API may be inferior to that utilized by Google. In cases in which sources of information other than Stack Overflow were returned the limitation is in the corpus itself as noted in Observation I, however the cases in which top results were also Stack Overflow articles appear to support this claim. An additional limitation of the search functionality provided by the Stack Overflow API is that it can not account for misspellings, meaning that if a user misspells a query term, it is likely that no results will be returned.

Observation III: There is a cold start problem to personalization - Finally, it was observed that the quality results returned by StackInTheFlow were of lower quality during the first half of the alloted time period than in the second. In the first half, if the issued query was generic, returned results may not have even related to Android, with ASP.NET results often returned. However, as the session continued, Android results were more consistently displayed in the top results.

This observation points to an issue in the result personalization approach utilized by StackInTheFlow. Until it has sufficient examples to ascertain which tags are relevant to the task at hand and which are not, barring manual selection of tags by the user, the returned results may be very generic based on the quality of the query. This was clearly observed in the case of Participant 3, who was forced to re-run a query as a result of Android Studio crashing. In the first instance of the query being run, non-Android results occupied the top rankings. However, in the second instance of the exact same query being executed Android results now occupied the top ranked results.

Unlike the previous two limitations, which could not be resolved without re-working core elements of StackInTheFlow. This issue may be overcome by introducing additional methods of inferring context from source code, an interesting direction for future research.

4.6 Threats to Validity

Within both evaluation methods there may be threats to validity, a selection of which is presented in the following section.

4.6.1 Threats to Construct Validity

In Evaluation Methodology I there are threats to construct validity in that reliance is made on manual queries as a basis for evaluating the effectiveness of the other query types. Ideally, all queries should be compared against an external baseline such as Google. In addition, the usage of clicks on results as a measure of success is far from ideal, as even if a result was investigated further (clicked on) it still may not prove useful. Both of these issues are addressed by Evaluation Methodology II, as the actual outcome of each query may be observed.

In Evaluation Methodology II, threats to construct validity come in the form of asking users to rate the performance of the tool on a 5 point Likert scale, and how

it was determined that a session was successful. Considering the former, such scales are a standard method of collecting participant feedback. Considering the latter, two separate annotators were utilized to limit individual biases and subjectiveness.

4.6.2 Threats to Internal Validity

In Evaluation Methodology I a threat to internal validity comes from a lack of knowledge regarding the experience and problem domains of the users of the tool. However, such lack of knowledge is necessary for the manner in which data was procured. In addition, with a sufficiently large sample size, biases toward a particular level of experience or problem domain may be mitigated.

In Evaluation Methodology II a threat to internal validity comes from the varying levels of experience of the study participants. To mitigate this, pre-study questionnaires asking the participants to asses this experience were collected so that it may be taken into account when considering the results.

4.6.3 Threats to Conclusion Validity

In Evaluation Methodology I, the raw frequency of each query type is reported in addition to their percentage makeup when calculating click ratios.

In Evaluation Methodology II, the raw frequency of each annotation is reported along with participant responses. The objective was to gain qualitative insight into the usefulness of StackInTheFlow, rather than to observe statistically significant results.

4.6.4 Threats to External Validity

Evaluation Methodology I represents a moderate sampling of developers. Though without prior knowledge as to their background and problem domain it cannot be excluded that they are biased in some way.

Evaluation Methodology II is a relativity small sampling of students with similar

experience levels. In addition the utilization of only one development task may greatly affect the results. Additional replication studies with a larger population of participants of varying experience levels and backgrounds in addition to multiple problem tasks would be desirable.

CHAPTER 5

CONCLUSIONS AND FUTURE RESEARCH

This thesis presents StackInTheFlow, a recommendation system for software engineering which integrates within the IntelliJ family of IDEs with the objective of automating the process of recommending Stack Overflow articles relevant to the software engineering task at hand. It also introduces and formalizes the concept of a Behavior-Driven Recommendation System for Software Engineering (BDRSSE), a recommendation system for software engineering that utilizes developer behavior to inform when recommendations should be made without any direct input from the developer. In other words, one which passively observes developer activity and makes recommendations when it is sufficiently confident such recommendations would be useful.

Two distinct issues to be addressed by a successful BDRSSE are also identified. That of determining when a recommendation should be made and, once such a determination has taken place, generating a query to retrieve relevant information without any direct input from the developer.

StackInTheFlow is one such recommendation system that achieves this. It does this by utilizing events extracted from the development logs of user activity to determine when the developer is in need of a recommendation. Specifically, it utilizes a ratio method between events deemed indicators of progress, such as the creation of new lines of code, to events deemed indicators of developer difficulty such as deleting lines of code and navigating through source code without making any edits. If the ratio of progress events to difficulty events drops below a certain threshold, a query is issued. Once it has been determined that a recommendation should be made, candidate terms are extracted from the currently open source code file within the IDE and scored against a term dictionary constructed from a data dump of Stack Overflow articles.

This dictionary contains various term quality metrics. From this, an overall score for each term is computed and the highest ranking terms are selected to form the query. In the case of a run or compile time error message, a similar process is utilized to form a query from the error stacktrace. StackInTheFlow also allows the user to manually specify a query as in the case of a typical search engine or to manually invoke the automatic query generation process.

StackInTheFlow also personalizes the results displayed to the user based on their past activity. It does this by utilizing a content-based filtering approach based around a novel metric Click Frequency-Inverse Document Frequency. It uses this metric to identify Stack Overflow article tags that are of interest to the user and to display results containing those tags higher in the ranking of results.

Two separate evaluation methodologies are utilized to asses the usefulness of StackInTheFlow. In the first, anonymous logs are collected from the users of StackInTheFlow to gauge which types of queries generated the most user interactions (clicks). The second method is a small study in which undergraduate students were given an Android development task with access to the tool. While completing the task, the student's screens are recorded. Later these recordings are annotated and combined with the results of a post-study questionnaire to asses the tool's usefulness.

Results indicate that manual queries performed with the tool were reasonably effective, however results were mixed for other query methods. Several limitations of the tool are identified based on the recordings extracted in the second evaluation method. Namely, that the information contained within the corpus of Stack Overflow is limited compared to that available to external search engines such as Google and that the search algorithm provided by the Stack Overflow API and utilized by StackInTheFlow appears to produce lower quality results when compared to the algorithm utilized by Google. In addition, a cold start problem is identified in regards to the relevancy of results initially returned by the tool. Enhancing the result personalization facilities of

StackInTheFlow may be a way to mitigate this issue.

Future research may focus on fine-tuning the method in which the various term quality metrics utilized in query formulation are combined based on each individual developer, as currently they are just linearly summed without weighting. In addition the adjustment of the difficulty detection thresholds may also be examined and fine-tuned on an individual basis. With methods of automatically adjusting these thresholds being a potentially fruitful area of future research. Enhancements to the manner in which candidate query terms are extracted from source code may also be examined. Currently a simple bag-of-words model is utilized, however better results may be achievable with more sophisticated methods such as "island" parsers. Perhaps the most interesting and potentially rewarding area of future research is in enhancing methods of inferring developer context from source code as a method of handling the cold start problem to result personalization. Finally, several quality of life enhancements to the tool may be considered such as porting it to additional platforms and languages as well as enabling the search feature to handle misspellings.

REFERENCES

[1] Andrew J. Ko, Robert DeLine, and Gina Venolia. "Information Needs in Collocated Software Development Teams". In: *Proceedings of the 29th International Conference on Software Engineering*. 2007, pp. 344–353.

[2] Thomas D. LaToza, Gina Venolia, and Robert DeLine. "Maintaining Mental Models: A Study of Developer Work Habits". In: *Proceedings of the 28th International Conference on Software Engineering*. New York, NY, USA: ACM, 2006, pp. 492–501.

[3] Larry L. Constantine. *Constantine on Peopleware*. Prentice Hall, 1995.

[4] Janice Singer, Timothy Lethbridge, Norman Vinson, and Nicolas Anquetil. "An Examination of Software Engineering Work Practices". In: *CASCON First Decade High Impact Papers*. Riverton, NJ, USA: IBM Corp., 2010, pp. 174–188.

[5] Joel Brandt, Philip J. Guo, Joel Lewenstein, Mira Dontcheva, and Scott R. Klemmer. "Two Studies of Opportunistic Programming: Interleaving Web Foraging, Learning, and Writing Code". In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. 2009, pp. 1589–1598.

[6] Alberto Bacchelli, Tommaso Dal Sasso, Marco D'Ambros, and Michele Lanza. "Content Classification of Development Emails". In: *Proceedings of the 34th International Conference on Software Engineering*. IEEE Press. 2012, pp. 375–385.

[7] John Anvik, Lyndon Hiew, and Gail C. Murphy. "Who Should Fix this Bug?" In: *Proceedings of the 28th International Conference on Software Engineering*. ACM. 2006, pp. 361–370.

[8] Joel Brandt, Philip J. Guo, Joel Lewenstein, and Scott R. Klemmer. "Opportunistic Programming: How Rapid Ideation and Prototyping Occur in Practice".

In: *Proceedings of the 4th International Workshop on End-User Software Engineering*. 2008.

[9] Renee McCauley, Sue Fitzgerald, Gary Lewandowski, Laurie Murphy, Beth Simon, Lynda Thomas, and Carol Zander. "Debugging: A Review of the Literature from an Educational Perspective". In: *Computer Science Education* 18.2 (2008), pp. 67–92.

[10] Stack Overflow Developer Survey. *https://insights.stackoverflow.com/survey/2017*. 2017.

[11] Martin P. Robillard, Walid Maalej, Robert J. Walker, and Thomas Zimmermann. *Recommendation Systems in Software Engineering*. Springer Science & Business, 2014.

[12] Luca Ponzanelli, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, and Michele Lanza. "Mining StackOverflow to Turn the IDE into a Self-Confident Programming Prompter". In: *Proceedings of the 11th Working Conference on Mining Software Repositories*. 2014, pp. 102–111.

[13] Luca Ponzanelli, Alberto Bacchelli, and Michele Lanza. "Seahawk: Stack Overflow in the IDE". In: *Proceedings of the 2013 International Conference on Software Engineering*. 2013, pp. 1295–1298.

[14] Thanh Nguyen, Peter C. Rigby, Anh T. Nguyen, Mark Karanfil, and Tien N. Nguyen. "T2API: Synthesizing API Code Usage Templates from English Texts with Statistical Translation". In: *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 2016, pp. 1013–1017.

[15] Brock A. Campbell and Christoph Treude. "NLP2Code: Code Snippet Content Assist via Natural Language Tasks". In: *Proceedings of the 2017 International Conference on Software Maintenance and Evolution*. 2017.

[16] Christopher S. Corley, Federico Lois, and Sebastián Quezada. “Web Usage Patterns of Developers”. In: *Proceedings of the 2015 IEEE International Conference on Software Maintenance and Evolution*. 2015, pp. 381–390.

[17] Dietmar Jannach, Markus Zanker, Alexander Felfernig, and Gerhard Friedrich. *Recommender Systems: An Introduction*. Cambridge University Press, 2010.

[18] Martin Szomszor, Ciro Cattuto, Harith Alani, Kieron O’Hara, Andrea Baldassarri, Vittorio Loreto, and Vito D.P. Servedio. “Folksonomies, the Semantic Web, and Movie Recommendation”. In: *Proceedings in the 4th European Semantic Web Conference, Bridging the Gap between Semantic Web and Web 2.0*. 2007.

[19] Greg Linden, Brent Smith, and Jeremy York. “Amazon.com Recommendations: Item-to-item Collaborative Filtering”. In: *IEEE Internet computing* 7.1 (2003), pp. 76–80.

[20] Moon-Hee Park, Jin-Hyuk Hong, and Sung-Bae Cho. “Location-based Recommendation System using Bayesian User’s Preference Model in Mobile Devices”. In: *Proceedings of the International Conference on Ubiquitous Intelligence and Computing*. Springer. 2007, pp. 1130–1139.

[21] Frank Mccarey, Mel Ó Cinnéide, and Nicholas Kushmerick. “Rascal: A Recommender Agent for Agile Reuse”. In: *Artificial Intelligence Review* 24.3-4 (2005), pp. 253–276.

[22] Oliver Hummel, Werner Janjic, and Colin Atkinson. “Code Conjurer: Pulling Reusable Software Out of Thin Air”. In: *IEEE software* 25.5 (2008).

[23] Reid Holmes, Robert J Walker, and Gail C. Murphy. “Approximate Structural Context Matching: An Approach to Recommend Relevant Examples”. In: *IEEE Transactions on Software Engineering* 32.12 (2006), pp. 952–970.

[24] Anupriya Ankolekar, Katia Sycara, James Herbsleb, Robert Kraut, and Chris Welty. “Supporting Online Problem-Solving Communities with the Semantic Web”. In: *Proceedings of the 15th International Conference on World Wide Web*. ACM. 2006, pp. 575–584.

[25] Mik Kersten and Gail C. Murphy. “Using Task Context to Improve Programmer Productivity”. In: *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM. 2006, pp. 1–11.

[26] Hans-Jörg Happel and Walid Maalej. “Potentials and Challenges of Recommendation Systems for Software Development”. In: *Proceedings of the 2008 International Workshop on Recommendation Systems for Software Engineering*. ACM. 2008, pp. 11–15.

[27] Francesco Ricci, Lior Rokach, and Bracha Shapira. “Introduction to Recommender Systems Handbook”. In: *Recommender Systems Handbook*. Springer, 2011, pp. 1–35.

[28] Yida Tao, Yingnong Dang, Tao Xie, Dongmei Zhang, and Sunghun Kim. “How Do Software Engineers Understand Code Changes?: An Exploratory Study in Industry”. In: *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. ACM. 2012, p. 51.

[29] Michael D. Ekstrand, John T. Riedl, and Joseph A. Konstan. “Collaborative Filtering Recommender Systems”. In: *Foundations and Trends® in Human–Computer Interaction* 4.2 (2011), pp. 81–173.

[30] J. Ben Schafer, Dan Frankowski, Jon Herlocker, and Shilad Sen. “Collaborative Filtering Recommender Systems”. In: *The Adaptive Web*. Springer, 2007, pp. 291–324.

[31] Gábor Takács, István Pilászy, Bottyán Németh, and Domonkos Tikk. “Scalable Collaborative Filtering Approaches for Large Recommender Systems”. In: *Journal of Machine Learning Research* 10.Mar (2009), pp. 623–656.

[32] Michael J. Pazzani and Daniel Billsus. “Content-Based Recommendation Systems”. In: *The Adaptive Web*. Springer, 2007, pp. 325–341.

[33] Robin Van Meteren and Maarten Van Someren. “Using Content-Based Filtering for Recommendation”. In: *Proceedings of the Machine Learning in the New Information Age: MLnet/ECML2000 Workshop*. 2000, pp. 47–56.

[34] Shari Trewin. “Knowledge-Based Recommender Systems”. In: *Encyclopedia of Library and Information Science* 69.Supplement 32 (2000), p. 180.

[35] Derek Bridge, Mehmet H Göker, Lorraine McGinty, and Barry Smyth. “Case-Based Recommender Systems”. In: *The Knowledge Engineering Review* 20.3 (2005), pp. 315–320.

[36] Brendon Towle and Clark Quinn. “Knowledge Based Recommender Systems Using Explicit User Models”. In: *Proceedings of the AAAI Workshop on Knowledge-Based Electronic Markets*. 2000, pp. 74–77.

[37] Gloria Mark, Shamsi T. Iqbal, Mary Czerwinski, and Paul Johns. “Bored Mondays and Focused Afternoons: The Rhythm of Attention and Online Activity in the Workplace”. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM. 2014, pp. 3025–3034.

[38] Davor Cubranic, Gail C. Murphy, Janice Singer, and Kellogg S. Booth. “Hipikat: A Project Memory for Software Development”. In: *IEEE Transactions on Software Engineering* 31.6 (2005), pp. 446–465.

[39] Alexey Zagalsky, Ohad Barzilay, and Amiram Yehudai. “Example Overflow: Using Social Media for Code Recommendation”. In: *Proceedings of the Third Inter-*

national Workshop on Recommendation Systems for Software Engineering. IEEE Press. 2012, pp. 38–42.

[40] Erik Hatcher and Otis Gospodnetic. "Lucene in Action". In: (2004).

[41] Christoph Treude, Mathieu Sicard, Marc Klocke, and Martin Robillard. "TaskNav: Task-Based Navigation of Software Documentation". In: *Proceedings of the 37th International Conference on Software Engineering*. Vol. 2. IEEE Press. 2015, pp. 649–652.

[42] Laurie Williams. "Integrating Pair Programming into a Software Development Process". In: *Proceedings of the 14th Conference on Software Engineering Education and Training*. IEEE. 2001, pp. 27–36.

[43] Ekwa Duala-Ekoko and Martin P. Robillard. "Asking and Answering Questions about Unfamiliar APIs: An Exploratory Study". In: *Proceedings of the 34th International Conference on Software Engineering*. IEEE Press. 2012, pp. 266–276.

[44] Ioanna Katidioti, Jelmer P. Borst, Marieke K. van Vugt, and Niels A. Taatgen. "Interrupt Ee: External Interruptions are Less Disruptive than Self-Interruptions". In: *Computers in Human Behavior* 63 (2016), pp. 906–915.

[45] Ashish Kapoor, Winslow Burleson, and Rosalind W. Picard. "Automatic Prediction of Frustration". In: *International Journal of Human-Computer Studies* 65.8 (2007), pp. 724–736.

[46] Sebastian C. Müller and Thomas Fritz. "Stuck and Frustrated or in Flow and Happy: Sensing Developers' Emotions and Progress". In: *Proceedings of the 37th International Conference on Software Engineering*. Vol. 1. IEEE. 2015, pp. 688–699.

[47] James Fogarty, Andrew J. Ko, Htet H. Aung, Elspeth Golden, Karen P. Tang, and Scott E. Hudson. "Examining Task Engagement in Sensor-Based Statistical

Models of Human Interruptibility". In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM. 2005, pp. 331–340.

[48] Rahul Nair, Stephen Voida, and Elizabeth D. Mynatt. "Frequency-Based Detection of Task Switches". In: *Proceedings of the 19th British HCI Group Annual Conference*. Vol. 2. 2005, pp. 94–99.

[49] Jason Carter and Prasun Dewan. "Design, Implementation, and Evaluation of an Approach for Determining when Programmers Are Having Difficulty". In: *Proceedings of the 16th ACM International Conference on Supporting Group Work*. 2010, pp. 215–224.

[50] Manuela Züger, Christopher Corley, André N. Meyer, Boyang Li, Thomas Fritz, David Shepherd, Vinay Augustine, Patrick Francis, Nicholas Kraft, and Will Snipes. "Reducing Interruptions at Work: A Large-Scale Field Study of Flowlight". In: *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*. ACM. 2017, pp. 61–72.

[51] Luca Ponzanelli, Alberto Bacchelli, and Michele Lanza. "Leveraging Crowd Knowledge for Software Comprehension and Development". In: *Proceedings of the 17th European Conference on Software Maintenance and Reengineering*. IEEE. 2013, pp. 57–66.

[52] Thomas M. Cover and Joy A. Thomas. *Elements of Information Theory*. John Wiley & Sons, 2012.

[53] Vladimir I. Levenshtein. "Binary Codes Capable of Correcting Deletions, Insertions, and Reversals". In: *Soviet Physics Doklady*. Vol. 10. 8. 1966, pp. 707–710.

[54] David Carmel and Elad Yom-Tov. *Estimating the Query Difficulty for Information Retrieval*. Morgan & Claypool, 2010.

[55] Ying Zhao, Falk Scholer, and Yohannes Tsegay. “Effective Pre-Retrieval Query Performance Prediction Using Similarity and Variability Evidence”. In: *Proceedings of the 2008 European Conference on Information Retrieval*. Springer. 2008, pp. 52–64.

[56] Chris Buckley. “Automatic Query Expansion Using SMART : TREC 3”. In: *Proceedings of the Third Text REtrieval Conference*. 1994, pp. 69–80.

VITA

Chase Greco was born on September 10th, 1995 in Bellingham, Washington. In 2017 he received his Bachelor's of Science in Computer Science from Virginia Commonwealth University, Richmond, Virginia. He is currently working towards his Master's Degree in Computer Science also at Virginia Commonwealth University. Greco's research is targeted towards recommendation systems, with a specialization in developing systems which support software engineers.