What to Learn Next: Recommending Commands in a Feature-rich Environment

by

Sedigheh Zolaktaf

B.Sc., Sharif University of Technology, 2013

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF

Master of Science

in

The Faculty of Graduate and Postdoctoral Studies
(Computer Science)

THE UNIVERSITY OF BRITISH COLUMBIA
(Vancouver)

August 2015

Abstract

Despite an abundance of commands to make tasks easier to perform, the users of feature-rich applications, such as development environments and AutoCAD applications, use only a fraction of the commands available due to a lack of awareness of the existence of many commands. Earlier work has shown that command recommendation can improve the usage of a range of commands available within such applications.

In this thesis, we address the command recommendation problem, in which, given the command usage history of a set of users, the objective is to predict a command that is likely useful for the user to learn. We investigate two approaches to address the problem.

The first approach is built upon the hypothesis that users of feature-rich applications who have similar features tend to use the same commands, and also, a specific user tends to use commands with similar features. Building on this hypothesis, we describe a supervised learning framework that exploits features from a user-command network to predict new links among users and commands.

The second approach is built upon three hypotheses. First, we hypothesize that in feature-rich applications there exists *co-occurrence patterns* between commands. Second, we hypothesize that users of feature-rich applications have prevalent *discovery patterns*. Finally, we hypothesize that users need different recommendations based on the *time elapsed* between their last activity and the time of recommendation. To generate recommendations, we obtain co-occurrence and discovery patterns from the command usage history of a large set of users of the same feature-rich application. Subsequently, for each user, we produce recommendations based on the user's command usage history, co-occurrence and discovery patterns, and time elapsed since the last command usage. We

refer to the algorithm we developed according to this approach as *CoDis*.

Empirical experiments on data submitted by users of an integrated development environment (Eclipse) demonstrate that *CoDis* achieves significant performance improvements over link prediction, standard algorithms used for command recommendation, and matrix factorization techniques that are known to perform well in other domains. Compared to ADAGRAD, the best performing baseline, it achieves an improvement of 10.22% in recall, for a top-N recommendation task ($N = 20$).

Preface

This thesis is submitted in partial fulfilment of the requirements for a Master of Science Degree in Computer Science. All the work presented in this dissertation are original and independent work of the author, performed under the supervision of Prof. Gail C. Murphy.

Table of Contents

List of Tables

List of Figures

Acknowledgments

I would like to thank my supervisor, Prof. Gail C. Murphy, for all her support, help, and encouragements. I am sincerely grateful for the kind guidance she provided, which allowed me to develop this thesis.

I would like to thank Prof. Laks V.S. Lakshmanan for being the second reader of my thesis and for providing helpful comments.

I would also like to thank my sister, Zainab, for being the third (informal) reader of my thesis and also for making life much more fun throughout my masters.

Lastly, I would like to thank my family, for their love, their belief in me, and their never-ending encouragement.

Dedication

To my family.

Chapter 1

Introduction

1.1 Motivation

Integrated development environments (IDE) provide comprehensive tools and facilities to assist programmers with software development tasks. In particular, these tools support various phases of the software development cycle. For instance, source code editors integrated in these environments are designed to simplify and speed up the input of source code, while debugging tools make the identification and fixing of bugs easier. To use any of these tools, the user is required to execute commands either through a command line, a graphical user interface or a keyboard short cut. Given the complexity of developing code, these environments quickly become feature-rich and complex, thereby making them less effective than aspired. For instance, consider the Eclipse integrated development environment[1]. Logs collected from a large set of anonymous users of this IDE with the Eclipse Usage Data Collector (UDC)[2][3] in 2009, show that 1098 distinct commands are used across 897714 users, but the majority of the users utilize less than 10 commands solely (Figure 1.1a). Even the majority of users who were continuously active during the entire year consisting of 4308 users, use less than 60 distinct command individually, whereas altogether they use 700 commands (Figure 1.1b).

This lack of awareness of commands has also been reported for other highly-functional applications, such as AutoCAD [27, 37], a computer-aided drawing application. For development environments, which have an ability to be easily extended with new tools and hence new commands, the problem is exacerbated. The efficiency of a user in performing tasks with these applications can be increased by making the users aware of commands relevant to them.

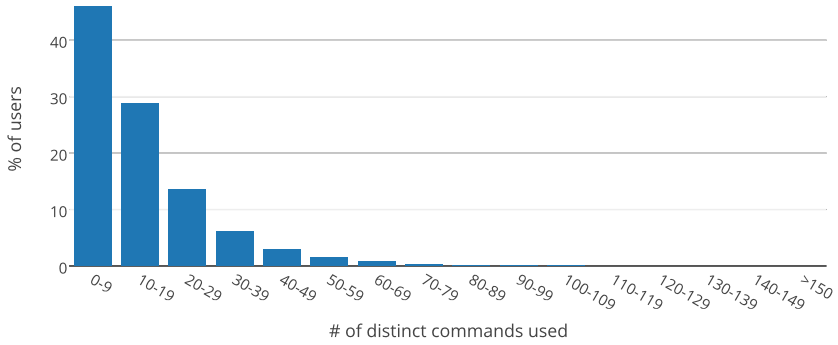
[1] https://www.eclipse.org/

[2] https://github.com/DeveloperLiberationFront/UsageDataCollectorOnBigData

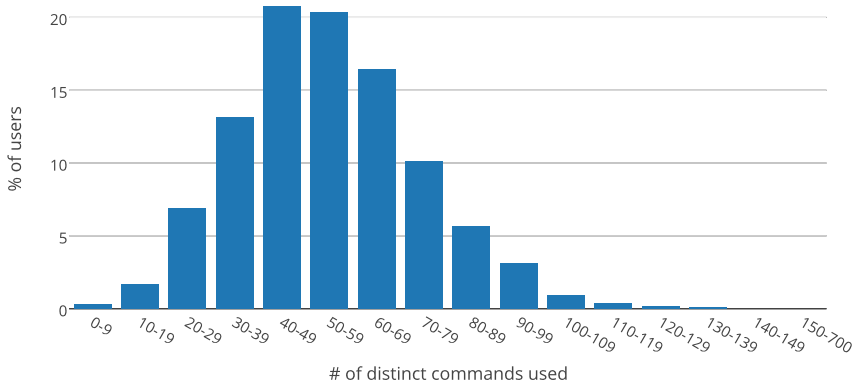
[3] https://www.eclipse.org/org/usagedata/

One approach for increasing a user's awareness of features in a feature-rich application is by providing efficient search mechanisms. This approach requires thorough documentation of the application, which must adjust with changes to the application over time. Even more critically, if a user is unaware of the existence of a feature, it will be difficult, if not impossible, to search for commands supporting the feature.

An alternative approach is through personalized command recommendation based on command usage history, which we call the command recommendation problem.



(a) Command usage distribution of all users.



(b) Command usage distribution of users who were active during an entire year.

Figure 1.1: Command usage distribution of Eclipse users obtained by UDC.

1.2 Problem Statement

In the command recommendation problem, given the command usage history of a set of users, the objective is to predict a command that will likely be useful for the user to learn. The command usage history of a user consists of a time-ordered sequence of commands issued by the user for the feature-rich application. We denote the set of users as $\mathscr{U}$ and the set of all commands issued by the users as $\mathscr{I}$.

To obtain recommendation lists, for each user and unused command pair (u, i), where $u \in \mathscr{U}$ and $i \in \mathscr{I}$, we compute a score that represents how useful command i would be for the user u. Subsequently, we recommend the top N unknown commands with the highest score to the user.

In this thesis, we focus on command recommendations for IDEs. We investigate two approaches to address the command recommendation problem: (1) link prediction as a supervised learning task and (2) an algorithm that we name *CoDis*.

1.3 Link Prediction

Link prediction [29] has been used for recommendation in different domains, such as co-authorship networks [2, 53] and e-commerce [28]. A user-item recommendation problem can be formalized as a link prediction problem by representing user and items as nodes in a bipartite graph and representing user-item interactions as links between user-item nodes. Link prediction deals with recommendation by trying to infer which new links between the user and item vertices are likely to occur in the graph, given a snapshot of the bipartite graph.

If existing links have similar features that distinguish them from non-existent links, link prediction can be solved using a supervised learning framework. In the command domain, we assume users who have similar features tend to use the same commands, and also, a specific user tends to use commands with similar features. Based on this assumption, we convert the command recommendation problem to a link prediction problem. We model users and commands as a heterogeneous bipartite graph, where nodes represent users and commands, and links represent user command interactions. We build upon the work of Huang and colleagues [21] to extract topological similarity metrics from a user-command bipartite graph. We then apply supervised learning algorithms to infer for each user, which commands they are likely to form an link with and generate recommendations based on the predicted links.

1.4 CoDis

Previous work has addressed the command recommendation problem through collaborative filtering techniques. Li, Matejka, and colleagues [27, 37] use user-based and item-based collaborative filtering to generate personalized command recommendations in AutoCAD. Murphy-Hill and col-

leagues [44] use collaborative filtering techniques based on sequential pattern mining [52] and patterns of past command discovery, to recommend commands in Eclipse. In this work, we build upon similar assumptions as [44] and [52] to improve the task of command recommendation.

In most domains, such as that of text collections or electronic retail, there are sets of frequently co-occurring items [3, 19, 46, 52]. This phenomenon is even more noticeable in the domain of IDE commands, where certain sets of commands have to be used to complete specific tasks [46, 56]. Hence, based on the hypothesis that in an IDE, frequent command *co-occurrence patterns* exist, we estimate co-occurrence scores for each user and command (*CScore* algorithm). We apply a variant of bigram models [8, 39] to compute this score. These models make predictions using observed marginal and conditional item frequencies.

Murphy-Hill and colleagues [44] also showed that there exists prevalent command *discovery patterns* in the domain of IDE commands, and that these patterns can be incorporated into collaborative filtering algorithms to make useful recommendations. Based on the intuition that in an IDE, frequent command discovery patterns exist, we estimate discovery scores for each command and user (*DScore* algorithm). Similar to co-occurrence scores, we use bigram models to compute discovery scores.

Finally, users have different needs in different circumstances. We believe the *elapsed time* between the user's last activity and the time of recommendation is an indicator of the user's recommendation need. In particular, if a user's elapsed time is relatively small, they are more likely to be working on their most recent tasks. However, if the elapsed time is relatively large, the user is more likely to have begun a new task. Based on this intuition, our solution consists of combining the co-occurrence and discovery score of each user with regard to their elapsed time to make recommendations. We define a tuning parameter λ to control the influence of co-occurrence and discovery scores. If the users elapsed time is less than λ, then the user is more likely to be working on their most recent task and recommendation should take into account their co-occurrence score more than their discovery score. However, if the elapsed time is more than λ, then recommendation should take into account their discovery score more than their co-occurrence score. We refer to the algorithm we developed according to this approach as *CoDis*.

1.5 Thesis Contributions

The contributions of this thesis include:

- We formulate the command recommendation problem as a supervised learning link prediction problem. We show how we can extract features from command usage history of users, described in Chapter 3. Furthermore, we describe how we can extract positive and negative examples from command usage history of users, and how to generate recommendation lists

from predicted links, described in Chapter 5.

- We propose an algorithm *CoDis* to generate command recommendations with regard to the elapsed time between users' last activity time and time of recommendation. The *CoDis* algorithm is described in Chapter 4.

- We conduct empirical experiments on data submitted by users of an integrated development environment (Eclipse) to compare the link prediction approach and the *CoDis* algorithm with standard algorithms used for command recommendation and matrix factorization techniques that are known to perform well in other domains.

1.6 Summary of Results

Empirical results on data collected by Eclipse Usage Data Collector demonstrate the effectiveness of the *CoDis* algorithm compared to link prediction, standard algorithms used for command recommendation, and matrix factorization techniques that are known to perform well in other domains. From among the link prediction algorithms, Random Forest obtains the highest accuracy. However, the *CoDis* algorithm beats all link prediction and baseline algorithms based on the recall measure. Compared to the matrix factorization technique with ADAGRAD solver [11], the best performing baseline, *CoDis* and Random Forest obtain an improvement of 10.22% and 4.74% in terms of the recall performance measure for a top-N recommendation task, where $N = 20$.

1.7 Notations and Conventions

Table 1.1 shows the notation we use in this work. We use lower case letters for scalar variables (e.g., a), upper case letters for constants (e.g., A), lower-case bold letters for vectors (e.g., $\mathbf{a}$), upper case bold letters for matrices (e.g., $\mathbf{A}$), and typeset the sets (e.g., $\mathscr{A}$).

Table 1.1: Notation

Variable	Symbol
Set of users	$\mathscr{U}$
Set of commands	$\mathscr{I}$
Set of neighbors of vertex v	$\mathscr{G}^v$
Set of neighbors of vertex v's neighbors	$\widehat{\mathscr{G}^v}$
Set of commands used (discovered) by user u	$\mathscr{I}^u$
Set of sessions of user u	$\mathscr{S}^u$
List of recommendations for user u	$\mathscr{R}^u$
Entire command usage history of the users	$\mathscr{Q}$
Command usage history of user u	$\mathscr{Q}^u$
Train set of user u	$\mathscr{Q}^u_{train}$
Test set of user u	$\mathscr{Q}^u_{test}$
Train set of user u in link prediction, consists of positive and negative examples	$\mathscr{A}^u_{train}$
Test set of user u in link prediction, consists of positive and negative examples	$\mathscr{A}^u_{test}$
Vector of feature weights	$\mathbf{w}$
Vector of features for instance *ins*	$\mathbf{x}_{ins}$
Session weights for user u	$\mathbf{w}^u$
Number of commands in a user's testing set using *k-tail*	k
Corresponding class of instance *ins*	y_{ins}
Elapsed time for user u	e^u
Edge weight between vertices u and c	$f^{u,c}$
Session chunk threshold	θ
Decay rate	λ
Number of recommendations	N
Inverse of regularization strength	C
Number of instances in link prediction training set	m
Command session matrix for user u	$\mathbf{N}^u$
Command co-occurrence frequency matrix	**CF**
Command co-occurrence probability matrix	**CP**
Command discovery frequency matrix	**DF**
Command discovery probability matrix	**DP**

1.8 Thesis Outline

The remainder of the paper is organized as follows: Chapter 2 presents a brief description of the most important research related to our work, Chapter 3 describes our investigation of link prediction, Chapter 4 describes our investigation of the CoDis algorithm, Chapter 5 reports on experiments performed, Chapter 6 outlines drawbacks and advantages of our approaches, and finally Chapter 7 concludes with a summary and a description of possible future directions.

Chapter 2

Related Work

In this chapter, we summarize related work for increasing users' awareness of commands in feature-rich applications. We also provide background on link prediction and discuss efforts relevant to the *CoDis* algorithm.

2.1 Increasing Users Awareness of Commands

Prior work on supporting users awareness of features provided by feature-rich applications can be classified in two dominant forms: tutorials and recommendation. We describe each of these bodies in the following subsections.

2.1.1 Tutorials for Application Usage

Users intentionally search through tutorials when they need assistance with a task. Prior work on web tutorial roles for feature-rich applications [26] has shown that users generally expect two types of help from tutorials: (1) some users expect to have help from tutorials in the midst of performing a task that they do not know how to complete, and (2) some users expect tutorials to pro-actively expand their skills. The latter users prefer to learn technology that could be used in later tasks, whereas the former users prefer to learn it when it is required. However, if a user is not aware of an existing feature they might never inquiry about it. Moreover, some applications continuously upgrade and provide more features, which makes it impossible for most users to know the existence of all the features provided. Thus, recommender systems are required to help increase users awareness of features. In this thesis, we extend the findings in [26] and conjecture users are more willing to accept in-task recommendation help when they are in the midst of performing a task (have a small elapsed time), and are more willing to accept recommendations that expand their repertoire of skills when they are not in the midst of performing a task (have a large elapsed time).

The *CoDis* recommendation algorithm we propose, produces different recommendations for users based on their elapsed time.

2.1.2 Recommendation Techniques for Application Usage

Human-to-Human Recommendation

One of the most influential ways to learn how to perform a software related task, is to learn from a peer performing the task [43]. This phenomenon is called Over the Shoulder Learning (OTSL) [55]. However, it has been shown that this method occurs less frequent than other methods for discovering new tools in feature rich environments [43]. This difference may be due to the fact that peers are not always available to help or may not know how to perform a task.

Murphy-Hill and colleagues [42] introduced Continuous Social Screencasting to overcome availability requirements in OTSL. This approach continuously and automatically records video clips of software developers working on tasks. The developers can share the recorded videos with other developers to demonstrate how to use a tool. However, this method does not overcome all availability requirements, since all software users will not have access to the videos if not shared with them. Moreover, Lubick and colleagues [35] conducted a field study that showed the learning effects were not as effective as OTSL.

Another effective method for learning tools from peers is remotely through online programming question and answer forums like Stack Overflow and question and answer forums embedded within interfaces like IP-QAT [38]. Through these communities, people have access to knowledge and expertise of their peers. However, even with this method, questions are not guaranteed to be answered on time. If an answer to a question does not exist, then the user would be required to post the question and might receive a late response. Even more critically, if users are unaware of the existence of an existing tool, they are unlikely to intentionally search for supporting tools.

Web Documentation Recommender Systems

Khan and colleagues [22] explored web documentation as data source to generate command recommendations for tasks at hand. They proposed a method that uses QF-Graphs [15], a bipartite graph which maps search queries (tasks) to commands (features) referenced in online web documentations. Based on a user's last x command usage, connected query nodes are activated based on their weights. This leads to estimating which queries are relevant to the user's current task. Then based on the activated queries, connected commands to the queries are consequently activated and used for command recommendation to the user. However, in [22], empirical experiments on data submitted by GIMP users, show that the proposed method achieved lower predictive accuracy then

previous work used in adaptive interfaces (e.g., frequency based predictions [13, 16]) for users with stable usage patterns.

Inefficiency-based Recommender Systems

Inefficiency-based recommenders make recommendations when they detect inefficient behaviour of a user performing a task. The Lumiere [20] project, which ultimately became Microsoft office '97 Assistant (Clippy), was one of the first inefficiency-based recommender systems for software tools. The program watched a user's activity using a tool and inferred whether they needed help. Lumiere used Bayesian models to reason under uncertainty and to infer likely intentions of users. It made several assumptions about inefficient performance, such as time spent on portions of a document, to obtain the model.

Another inefficiency recommender is Spyglass [56]. Spyglass aims to increase users awareness of navigations tools in IBM Rational Team Concert Environment which is based on Eclipse. Based on the assumptions that recent events contribute more to a developer's current goal and that selection of information elements may imply navigation, Spyglass tracks up to seven of the most recent navigation events in a window of recent activity. It uses the window of recent activity to infer when a developers selections may indicate a navigation between two pieces of information. It computes the proximity between the first event in the window and events farther. Then it compares the proximity number to the optimal number of information elements needed to reach each event using different tools. If a tool is found that may be a more efficient means to support navigation, it adds the tool to a set of potential recommendations. When the window of recent activity is full, it recommends all the unused tools from the set of potential recommendations to the user.

A limitation of the above recommender systems is the requirement of expert knowledge to precondition each command and activity for recommendation. Recommender systems which do not need pre-programming have also been explored. OWL [33] is a recommender system for Microsoft Word, which intends to enhance the organization-wide learning of application software. OWL observes the command usage of a large number of users over a period of time. It makes recommendations to individuals to change their command usage behaviour by comparing a user's command usage to the average command usage across the organization. The system would then make a recommendation if a command is being under-utilized or over utilized by an individual in comparison to the organization.

A common drawback of the aforementioned inefficiency-based recommenders, is that they are not personalized for individuals. Moreover, even though users knowledge of tool changes over time the same models are repeatedly used.

Neighborhood Collaborative Filtering Recommender Systems

In [27, 37, 44], collaborative filtering methods have been investigated in software environments to increase users awareness of environment commands. These models address the task of IDE command recommendation by analysing implicit user-command interactions to detect common command usage patterns. Li, Matejka and colleagues [27, 37] introduce CommunityCommands, a recommender system for AutoCAD. This system records users' command usage history and adapts the tfidf [51] weighting scheme to commands. Then it applies user-based and item-based collaborative filtering to make a recommendation list for each user.

Murph-Hill and colleagues [44] build upon [27, 37] by modelling patterns of command discovery. They introduced the notion of discovery patterns in commands, which means that it is prevalent in the user community to discover some commands after others. They assumed if a user uses command i then discovers command j, then the discovery pattern (i, j) occurs. However, they had several constraints for a command being actually discovered. First of all, they assumed all commands being used in the first period of the users activity (base command usage window) are known to the user. Second, they ruled out commands which were used only once. They showed that using discovery patterns for command recommendation can be more useful than only considering their interactions, by applying collaborative filtering techniques to discovery patterns of users. They also presented a proof-of-concept study that proves the existence of prevalent discovery patterns in Eclipse. In this work, we make our second hypothesis on discovery patterns of commands based on [44] and we incorporate it with our first hypothesis about co-occurrence patterns. However, the the definition of discovery patterns that we use differs from the definition presented by [44].

2.2 Link Prediction

The task of link prediction is to infer which interactions among members of a network are likely to occur in the near future, given a snapshot of the network [29]. The two most common type of networks are (1) unimodal networks, which can be presented as a homogeneous graph, where nodes represent users and links represent user ties such as co-authorship, and (2) affiliation networks which can be presented as a heterogeneous bipartite network, where nodes represent users and items and links represent user and item ties such as purchases [60]. Two popular frameworks for solving link prediction problems include (1) unsupervised learning framework, and (2) supervised learning framework, where they both take advantage of topological features and non-topological features of users and items. Topological features are extracted from the structure of an input graph and include common metrics such as Adamic Adar and Common Neighbors [29]. Non-topological features depend on metadata and attributes of the users and items. For example in an co-authorship network[53], such attributes consist of keyword similarities, paper topic similarities, etc.

In unsupervised learning link prediction, connection scores are assigned to all pair of nodes, which are calculated based on the features. Then pairs are ranked according to their scores and presented to users for recommendation. For example, Libin-Nowell and colleagues [29] apply unsupervised learning methods to a co-authorship network. They explore common topological connection scores such as Adamic Adar, Common Neighbors, Kitz, and Shortest Path for ranking nodes and recommending potential collaborators to authors.

In supervised learning link prediction, node pairs are associated with a number of features and are typically classified into two classes using classification algorithms: existing links (positive class) and non-existing links (negative class). Given a new node pair and its corresponding features, the aim is to predict to which class it belongs. For example, Al Hasan and colleagues [2], apply a supervised learning framework to a unimodal co-authorship network. They extract features from metadata such as keywords in addition to topological features and apply different classification algorithms, such as Decision Tree[40], to predict collaborations between authors.

In this work, we use a similar approach to [21], where existing features such as Common Neighbors are revised to be meaningful to a bipartite graph, to extract features from a user-command network. It has been shown that supervised learning approaches out perform unsupervised learning approaches [30], which leads us to use supervised learning for the task of command recommendation in this thesis.

2.3 Prediction of Next Item

Predicting the next item a user will use or visit has been used in other domains, such as the News domain [17, 54]. Trevisiol and colleagues [54] showed that methods based on transition likelihoods can beat content based methods since users do not constantly read about the same topic. Garcin and colleagues [17] use the sequential nature of news reading to model it as a Markov process and generate recommendations based on transition probabilities.

In natural language processing tasks, N-Gram models have been widely used to predict the next word. For example, Brown et al. [8, 39] addressed the problem of predicting a word from previous words in a text by using frequency of co-occurrence of words in a document.

Our model uses bigram models in the command domain to obtain co-occurrence and discovery scores. However, our approach is different from previous work, because we consider co-occurrence and discovery patterns simultaneously. Moreover, we take into account the elapsed time between the users last activity time and time of recommendation as an indicator of their need and use it as an weighting scheme for the co-occurrence and discovery scores.

Chapter 3

A Link Prediction Approach

In this chapter, we investigate the use of link prediction to address the command recommendation problem. We focus on a description of the approach and delay the detailed evaluation to Chapter 5.

Our approach is built upon one hypothesis:

Hypothesis 1. *In an integrated development environment, users with similar features tend to use the same commands, and also, a specific user tends to use commands with similar features.*

Thereupon, user-command pairs in which the user has interacted with the command (positive instance) will have similar features, which distinguishes them from user-command pairs in which the user has not interacted with the command (negative instance). The positive instances belong to a positive class, whereas the negative instance belong to a negative class. Given a new user-command pair, we can predict in which class the user-command pair will belong to based on it's features.

From the command usage history of users, we can extract implicit topological features for users and commands based on user-command interactions. We map user-command interactions to a weighted bipartite user-command interaction graph, where the graph structure captures subtle information on relations between users and commands (Figure 3.1). We denote the bipartite user-command network as $\mathscr{G} = (U, I, F)$, where partitions U and I correspond to the user set $\mathscr{U}$ and command set $\mathscr{I}$ respectively. Also, F denotes the edges between the partitions, where edge $f^{u,i} \in F$ is the number of times user u uses command i. For user u, let $\mathscr{G}^u$ be the set of all commands which they use. We define $\widehat{\mathscr{G}^u} = \bigcap_{i \in \mathscr{G}^u} \mathscr{G}^i$ as the set of users who use a common command with user u. Similarly, for a command i, let $\mathscr{G}^i$ be the set of all users who use it. We define $\widehat{\mathscr{G}^i} = \bigcap_{u \in \mathscr{G}^i} \mathscr{G}^u$ to be the set of commands that are used by a common user with command i.

We extract meaningful features based on the structure of the graph and pose the command recommendation problem as a link prediction problem: Given a snapshot of a user-command network,

we aim to infer which new interactions among the user commands are likely to occur in the near future. We study link prediction as a supervised learning task to predict whether missing links between users and commands will form (positive instance) in the future or not (negative instance).

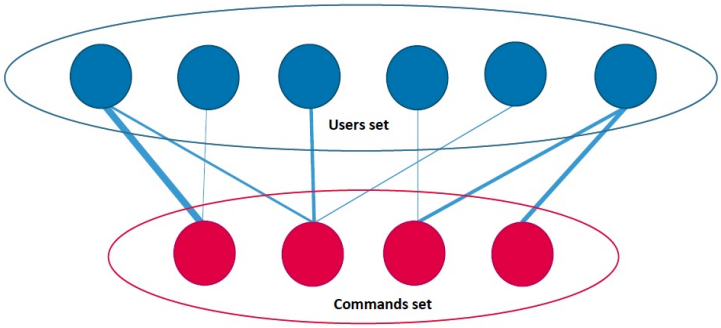


Figure 3.1: A representation of a weighted user-command bipartite graph, where edges represent user' interactions with commands.

3.1 Feature Set

Choosing an appropriate feature set is a critical part of any supervised learning algorithm. For each instance of user-command pair we use a number of features that are extracted from the command usage history of users. We use a similar approach to [21], to extract meaningful measures from a bipartite graph. Each feature relates either to the user node, command node, or both. In the following we provide a description of all the topological features we used for link prediction in the user-command network.

- **User Neighbors (UN)**: For user u, this feature measures the number of distinct commands that they use:

$$UN(u) = |\mathscr{G}^u| \tag{3.1}$$

- **Command Neighbors (CN)**: For command i, this feature measures the number of distinct users who use it:

$$CN(i) = |\mathscr{G}^i| \tag{3.2}$$

- **Preferential Attachment [21] (PA)**: For user u and command i, this feature relates the user's knowledge of commands with the command's popularity:

$$PA(u,i) = |\mathscr{G}^u| * |\mathscr{G}^i| \tag{3.3}$$

- **User Neighbors Weighted (UNW)**: For user u, this features measures the total number of times the user executes a command:

$$UNW(u) = \sum_{i \in \mathscr{G}^u} f^{u,i} \tag{3.4}$$

- **Command Neighbors Weighted (CNW)**: For command i, this feature measures the number of times it is used by the entire user community:

$$CNW(i) = \sum_{u \in \mathscr{G}^i} f^{u,i} \tag{3.5}$$

- **User Common Commands [21] (UCC)**: For user u and command i, this feature measures the number of commands user u uses in common with the users who use command i:

$$UCC(u,i) = |\mathscr{G}^u \cap \widehat{\mathscr{G}^i}| \tag{3.6}$$

- **Command Common Users [21] (CCU)**: For user u and command i, this feature measures the number of common users command i has with the commands used by user u:

$$CCU(u,i) = |\widehat{\mathscr{G}^u} \cap \mathscr{G}^i| \tag{3.7}$$

- **User Common Commands Jaccard's Coefficient [21] (UCCJ)**: For user u and command i, this feature measures the ratio of the number of commands user u uses in common with the users who use command i to the number of commands that are either used by user u or by the users who use command i:

$$UCCJ(u,i) = \frac{|\mathscr{G}^u \cap \widehat{\mathscr{G}^i}|}{|\mathscr{G}^u \cup \widehat{\mathscr{G}^i}|} \tag{3.8}$$

- **Command Common Users Jaccard's Coefficient [21] (CCUJ)**: For user u and command i, this feature measures the ratio of the number of common users command i has with the commands used by user u to the number of users who either use command i or a command used by user u:

$$CCUJ(u,i) = \frac{|\widehat{\mathscr{G}^u} \cap \mathscr{G}^i|}{|\widehat{\mathscr{G}^u} \cup \mathscr{G}^i|} \tag{3.9}$$

- **User Common Commands Adamic Adar [21] (UCCAA)** : This feature refines User Common Commands and considers common commands with a lower degree more important than

common commands with higher degree.

$$UCCAA(u,i) = \sum_{i' \in |\mathscr{G}^u \cup \widehat{\mathscr{G}^i}|} \frac{1}{log|\mathscr{G}^{i'}|} \tag{3.10}$$

- **Command Common Users Adamic Adar [21] (CCUAA)** : This feature refines Command Common Users and considers common users with a smaller degree more important than common users with higher degree:

$$CCUAA(u,i) = \sum_{u' \in |\widehat{\mathscr{G}^u} \cup \mathscr{G}^i|} \frac{1}{log|\mathscr{G}^{u'}|} \tag{3.11}$$

- **User Entropy [58] (UE)**: For user u, this feature measures the entropy of the user's interest in all commands. The broader a user's tastes and interests are, the more distinct commands they use. We assumed if a user uses a large number of commands especially with equal probability, the uncertainty of the user tends to increase. Inversely, if a user uses only a few commands with unequal probability, the specificity of the user tends to increase. Using information theory, command-based User Entropy of user u is defined as:

$$UE(u) = -\sum_{i \in \mathscr{G}^u} p(i/u) log p(i/u) \tag{3.12}$$

where $p(i/u) = \frac{f^{u,i}}{\sum_{i \in \mathscr{G}^u} f^{u,i}}$.

3.2 Classification Algorithms

There exist a proliferation of supervised learning classification algorithms. In this thesis, we use six classification algorithms to predict whether missing links in a user-command graph will form in the future or not. In the following, we give a brief description of the algorithms and assume m is the number of training instances, $\mathbf{w}$ is a vector consisting of feature weights, $\mathbf{x}_{ins}$ is a vector consisting of all features of instance *ins*, y_{ins} is the corresponding class of instance *ins*, and C is the inverse of regularization strength:

- **Decision Tree [40]**: uses simple decision rules inferred from data features, to recursively partition data. It has a multi stage approach, where at each stage in the procedure, a function such as information gain or Gini impurity [49] is used to process the data.

- **Random Forest [4]**: is a ensemble learning method that uses a collection of decision trees

for training. For an instance, each tree casts a vote on its class, then the random forest outputs the mode on the individual trees.

- **Logistic Regression [41]**: is a direct probability model for classification. As an optimization problem, a binary class L2 penalized logistic regression minimizes the following cost function:

$$min_{\mathbf{w},c} C \sum_{ins=1}^{m} log(exp(-y_{ins}(\mathbf{x}_{ins}^T \mathbf{w} + c)) + 1) + \frac{1}{2}||\mathbf{w}||^2 \tag{3.13}$$

- **K-Nearest Neighbors [7, 40]**: classifies instances based on the corresponding class of its closest K training instances in the features space. The algorithm uses a function such as inverse of Euclidean distance or Minkowski [7] metric to weight neighbors of each instance.

- **Support Vector Machine (SVM) [41]**: maps instances in space to classes, by widening a margin as a wide as possible between points of different classes. As an optimization problem, a linear binary class SVM minimizes the following cost function:

$$argmin_{\mathbf{w}} C \sum_{ins=1}^{m} max(0, 1 - y_{ins} \mathbf{w}^T \mathbf{x}_{ins}) + \frac{1}{2}||\mathbf{w}||^2 \tag{3.14}$$

  In addition to linear classification, SVMs can perform non-linear classification using kernel tricks such as a radial basis function (rbf) or a polynomial kernel.

- **Naïve Bayes [41]:** is based on Bayes' theorem with the naïve assumption of independence between every pair of features. The classifier uses the following function to assign positive and negative classes to instances:

$$y_{ins} = argmax_{class \in \{+,-\}} p(y_{class}) \prod_{ins=1}^{m} p(\mathbf{x}_{ins} | y_{class}) \tag{3.15}$$

  A Naïve Bayes classifier can use different likelihood functions such as the Gaussian distribution function or the Bernoulli distribution function.

We investigated Decision Trees because they are interpretable and easy to explain. Other algorithms such as Logistic Regression are not intuitive. However, decision trees can easily overfit the training data. Therefore, we considered an ensemble method such as Random Forest to avoid overfitting. We considered Logistic Regression because the model can easily be updated to take in new data unlike Random Forests and Decision Trees. Moreover, Logistic Regression inherently produces useful probabilities which can be be used for ranking recommendations. We investigated SVM because with a proper kernel it can work well even if the data is not linearly separable based

on the proposed features. However, SVMs are very slow to tune compared to other classification algorithms. We considered Naïve Bayes because it has linear time complexity compared to the size of the data and therefore has optimal time complexity. Also, Naïve Bayes decisions are interpretable.

Each classifier can predict a probability for a user-command pair forming a link (described in Section 5.3.1). For each user, we sort their incident edges based on the probabilities and make a top-N recommendation list of the corresponding unused commands.

Chapter 4

CoDis

In this chapter, we investigate our proposed algorithm, *CoDis*, to address the command recommendation problem. The detailed evaluation of the algorithm is explained in Chapter 5.

CoDis is built upon three hypotheses:

Hypothesis 2. *In an integrated development environment, frequent command co-occurrence patterns exist.*

Hypothesis 3. *In an integrated development environment, frequent command discovery patterns exist.*

Hypothesis 4. *If a user's elapsed time is relatively small, they are more likely to be working on their most recent work, and would prefer a recommendation based on co-occurring patterns. As their elapsed time increases, they are more likely to be working on a new task, and would therefore prefer a recommendation based on discovering patterns.*

To distinguish between co-occurrence and discovery patterns we divide a user's command usage history into chunks called sessions. To acquire sessions, we define a tunable session chunk threshold θ, which is determined by cross validation, and assume if the user resumes work after $t \geq \theta$, then they have started a new session. Otherwise, we assume that the user is carrying on with their previous session. We denote $\mathscr{S}^u$ as the set of sessions of user u.

In Sections 4.1 and 4.2, we describe how co-occurrence patterns and discovery patterns can be used to compute scores for user-command pairs, respectively. In Section 4.3, we propose a framework to combine the scores based on elapsed time of users. In Section 4.4, we combine the scores irrespective of elapsed time.

4.1 Co-occurrence Score (CScore)

To perform a task in software environments, the user needs to execute commands. However, commands can not be arbitrarily executed to complete a task; for each task a set of specific commands need to be executed. Additionally, there might be multiple ways to perform a single task. For example, for the task of moving text from one place to another in the Eclipse IDE, one user might issue the commands *cut* and *paste*, consecutively, whereas another user may instead execute the *copy*, *paste*, and *delete* commands. Thus, in a development environment, there are sets of frequently co-occurring commands:

Definition 1. [Command Co-occurrence Pattern] A command co-occurrence pattern (i, j) is defined as a user executing command i then j or j then i, sequentially in a single session. □

Hypothesis 2. *In an integrated development environment, frequent command co-occurrence patterns exist.*

We use a symmetric matrix $\mathbf{CF} \in \mathbb{R}^{|\mathscr{I}|\times|\mathscr{I}|}$ to record co-occurrence frequency of commands. Every time a pair of commands (i, j) are executed consecutively in a single session, their corresponding frequencies, i.e., $cf_{i,j}$ and $cf_{j,i}$, increase by one each. For example, if a user executes the commands i,j,k,i, and j sequentially in a single session, $cf_{i,j}$ and $cf_{j,i}$ increase by two, while $cf_{j,k}$, $cf_{k,j}$, $cf_{k,i}$, and $cf_{i,k}$ increase by one. We then use these co-occurrence frequencies to obtain the co-occurrence probability of a command j occurring with command i in a session. We record these probabilities in the command co-occurrence probability matrix $\mathbf{CP} \in \mathbb{R}^{|\mathscr{I}|\times|\mathscr{I}|}$. Here, $cp_{i,j}$ represents the probability of command j occurring with command i:

$$cp_{i,j} = \frac{cf_{i,j}}{\sum_{k=1}^{|\mathscr{I}|} cf_{i,k}} \tag{4.1}$$

Note that this matrix is asymmetric.

Moreover, for user u, we use the command usage history of the user to derive a command session matrix $\mathbf{N}^u \in \mathbb{R}^{|\mathscr{S}^u|\times|\mathscr{I}|}$, where $n^u_{s,j}$ is the normalized frequency count of command j in session s for user u. Hence, we have:

$$\sum_{j=1}^{|\mathscr{I}|} n^u_{s,j} = 1 \tag{4.2}$$

A user's recent command usage is indicative of the task they are working on. Since the same command can be used for many tasks, we can not exactly determine what the user is working on. However, using their previous command usage history and the above co-occurrence probability matrix, we can estimate the commands they would most likely use to complete their task. Therefore, for any user u and unused command i, we estimate a co-occurrence score using the following

equation:

$$CScore(u,i) = \sum_{s=1}^{|\mathscr{S}^u|} w_s^u \sum_{j=1}^{|\mathscr{I}|} n_{s,j}^u cp_{j,i} \tag{4.3}$$

where $\mathbf{w}^u \in \mathbb{R}^{|\mathscr{S}^u|}$ is a weight vector that defines the importance of user u's sessions. In our setting, we assumed only the user's last session is important:

$$w_s^u = \begin{cases} 1 & s = |\mathscr{S}^u| \\ 0 & \text{otherwise} \end{cases} \tag{4.4}$$

4.2 Discovery Score (DScore)

When users begin using a development environment, they start by using specific commands, and as they become more fluent they discover more and more commands [44]. For example, in an IDE such as Eclipse, most people expect there exists a *save* command even when they have not yet worked with it. But as time passes, they start learning more commands. One might next learn a basic command such as *rename* and then a more sophisticated version control command such as *compareWithRevision* [44]. In this work, we define command discovery patterns in development environments as:

Definition 2. [Command Discovery Pattern] We assume users discover a command when they first execute the command. A command discovery pattern (i, j) happens when a user first discovers command i in a session and discovers command j consecutively in a later session, with no other command being discovered in between. □

Hypothesis 3. *In an integrated development environment, frequent command discovery patterns exist.*

We use an asymmetric matrix $\mathbf{DF} \in \mathbb{R}^{|\mathscr{I}|\times|\mathscr{I}|}$ to record the frequency of discovery patterns in the user community, where $df_{i,j}$ shows the frequency of the discovery pattern (i, j). Note (i, j) is a discovery pattern only when i and j are discovered consecutively and in two distinct sessions. The latter constraint is put to distinguish discovery patterns with co-occurrence patterns; two commands that are discovered in the same session are likely to be co-occurring. For example, consider a user who discovers commands i and j in their first session, reuses j in their second session, and reuses i and discovers k in their third session. Then only $df_{j,k}$ increases by one, because j and k were discovered consecutively in two distinct sessions. $df_{i,j}$ will not increase because i and j were discovered in the same session. Also, $df_{i,k}$ will not increase because they were not discovered consecutively. We then use **DF** to derive the command discovery probability matrix $\mathbf{DP} \in \mathbb{R}^{|\mathscr{I}|\times|\mathscr{I}|}$.

Note that this matrix is also asymmetric. Here, $dp_{i,j}$ represents the probability of command j being discovered after command i in the entire user community:

$$dp_{i,j} = \frac{df_{i,j}}{\sum_{k=1}^{|\mathscr{I}|} df_{i,k}} \tag{4.5}$$

For user u and unused command i, based on the user's command discovery patterns and the command discovery pattern probabilities of the entire user community, we use the following equation to obtain the probability of the user discovering the command:

$$DScore(u,i) = \frac{\sum_{j \in \mathscr{I}^u} dp_{j,i}}{|\mathscr{I}^u|} \tag{4.6}$$

where $\mathscr{I}^u$ is the set of discovered commands of user u. The more prevalent the discovery patterns (j,i) are in the command usage history of the entire user community, where j is any command in the set of discovered commands of user u, the higher $DScore(u,i)$ will be. In other words, if user u uses commands that are discovered frequently before command i by the entire user community, then $DScore(u,i)$ will be high compared to when the users uses commands that are not discovered frequently before command i by the entire user community.

4.3 Combining Co-occurrence and Discovery Scores Based on Elapsed Time (CoDis)

Users have different requirements in different circumstances. When a user is busy with a task, they prefer recommendations relevant to that specific task. However, if they are just about to begin a new task, they may prefer more general recommendations. Therefore, the user's current status should be considered when making recommendations. To achieve this, we first define the user's elapsed time:

Definition 3. [Elapsed time] For each user u, we define e^u to be the elapsed time between the user's last activity and the time a recommendation is generated. □

Hypothesis 4. *If a user's elapsed time is relatively small, they are more likely to be working on their most recent work, and would prefer a recommendation based on co-occurring patterns. As their elapsed time increases, they are more likely to be working on a new task, and would therefore prefer a recommendation based on discovering patterns.*

Using elapsed time as an indicator of the user's recommendation need, and relying on the aforementioned hypothesis, we compute the final score of a user-command pair (u,i) considering

both the co-occurrence score $CScore(u,i)$, and the discovery score $DScore(u,i)$. Specifically, we combine the scores using a linear model that gradually decays the influence of the co-occurrence score as e^u increases:

$$CDScore(u,i) = CScore(u,i) * f(e^u) + DScore(u,i) \tag{4.7}$$

We define $f(e^u) = \frac{\lambda}{e^u}$ where λ is a tuning parameter that controls the influence of *CScore* and is determined by cross validation. Based on this intuition, if the users elapsed time is less than λ, then *CoDis* generates recommendations that take into account *CScore* more than *DScore* and if the users elapsed time is more than λ, then it generates recommendations that take into account *DScore* more than *CScore*. As the elapsed time of the user increases, the effect of *CScore* decreases.

For each user u and unused command, we compute the above score, and take the top-N unused commands as the recommendation list for the user. We refer to this algorithm as the *CoDis* model throughout this work.

4.4 Combining Co-occurrence and Discovery Score Irrespective of Elapsed Time (Co+Dis)

In addition to *CoDis*, we derived another model *Co+Dis* where the score of a user-command pair is computed irrespective of their elapsed time:

$$C+DScore(u,i) = CScore(u,i) + DScore(u,i) \tag{4.8}$$

We derive this model to compare it's accuracy with *CoDis* in Chapter 5 and to show that weighting co-occurrence and discovery scores by elapsed time in *CoDis* is effective. We refer to this algorithm as *Co+Dis* throughout this work.

Chapter 5

Experiments

5.1 Data Description

To test the efficiency of our algorithm, we conducted offline experiments on data collected from Eclipse Usage Data Collector (UDC). This dataset contains information voluntarily submitted from about 1 million Eclipse users in 2009. It has more than 1.3 billion events, including command invocations which are tagged by users identifiers and timestamps. We filtered the dataset to consider only command invocations. This reduced the dataset to 897714 users who executed 1098 distinct commands a total of 315048551 times. Also, similar to [44], we chose to experiment only on users who were active during the entire year to increase the chances of obtaining actual discovery patterns. This led to a final dataset consisting of 4308 users who used 700 distinct commands a total of 22926392 times.

We used the Eclipse data set consisting of 4308 users and 700 commands to conduct empirical experiments. We used four different subsets of data consisting of 3 months, 6 months, 9 months of users recent command usage history, and their entire command usage history to see how recommendation accuracy, i.e., recall, changes if we discard older data. We refer to these subsets as subset (a), subset (b), subset (c), and subset (d), respectively. The *k-tail* approach (described in Section 5.2) discarded users in (a), (b), (c), and (d) who had not discovered more than one command during the corresponding time. This reduced the dataset to 4178 users and 480 commands in (a), 4292 users and 638 commands in (b), 4301 users and 676 commands in (c), and 4305 users and 700 commands in (d).

5.2 Training, Validation, and Testing Sets

We used the *k-tail* approach as described in [27, 37, 44] to split all users' command usage history into a training set and a testing set. For each user, the testing sequence contains the last k commands that the user has discovered. The training set contains all the commands the user has used, until the first command in the testing set was used. For example, assume the user has a command usage history consisting of $\mathscr{Q}^u = \{commit, rename, commit, openResource, synchronize, \text{and } rename\}$. Note, that the last command discovery is *synchronize*. Now, if $k = 1$, the testing set would contain $\mathscr{Q}^u_{test} = \{synchronize\}$ which is the last discovered command. The training set contains all the previous commands used before *synchronize*, and consists of $\mathscr{Q}^u_{train} = \{commit, rename, commit, openResource\}$. The last command, *rename*, has been discarded because it was used after the last command discovery. To enable comparisons to earlier work [27, 37, 44], we use $k = 1$. This value is reasonable as our goal is to teach commands one at a time to each user.

For algorithms that need tuning to avoid over fitting the data, we used a validation set. For all algorithms except for the classification algorithms, to obtain the validation set, we used the *k-tail* approach with $k = 1$ on the users' training set to further split it into a training set and a validation set.

The link prediction classifiers require negative examples in addition to positive examples. However, in the Eclipse dataset, user-command pairs are not explicitly classified. We assumed if a user has used a command, then the command was useful to them. Therefore, we classified a user-command pair as a positive example if the users uses the command at least once, and as a negative example otherwise. We partitioned all positive examples into two non overlapping sets: training set and testing set using the *k-tail* approach, which was described previously. To partition negative examples, user-command pairs with a missing link were randomly split into two sets with the constraint that for each user their training set contained the same number of positive and negative examples. Hence, the training set was balanced with the same number of positive and negative examples, whereas the testing set contained more negative examples than positive examples. For example, let $\mathscr{I} = \{commit, rename, extractMethod, openResource, synchronize, sync, showHistory, format, copy, paste\}$, and $\mathscr{Q}^u = \{commit, rename, commit, openResource, synchronize, \text{and } rename\}$. As explained above for user u, $\mathscr{Q}^u_{train} = \{commit, rename, commit, openResource\}$ and $\mathscr{Q}^u_{test} = \{synchronize\}$. We extract three positive instances $(u, commit)$, $(u, rename)$, and $(u, openResource)$ for the training set and one positive instance $(u, synchronize)$ for the testing set. The negative instances are extracted based on u's unused commands and consist of $(u, extractMethod)$, $(u, sync)$, $(u, paste)$, $(u, showHistory)$, $(u, format)$, and $(u, copy)$. We randomly split the negative examples into two the training set and testing set such that the training

set also contains three negative examples. A possible division could be $(u, sync)$, $(u, paste)$, and $(u, showHistory)$ as negative instances for the training set and $(u, extractMethod)$, $(u, copy)$, and $(u, format)$ as negative examples for the testing set. We demonstrate the training set and testing set obtained with this approach that consists of both positive and negative examples as $\mathscr{A}^u_{train}$ and $\mathscr{A}^u_{test}$.

For the classification algorithms, we used grid search [1] to obtain a validation set (described in Section 5.5).

5.3 Algorithms & Implementation

5.3.1 Link Prediction Algorithms

We programmed the link prediction approach using Python. We first extracted the features mentioned in Chapter 3 from the command usage history of users and then standardized all the features using scikit-learn[2] [45], a machine learning library for Python. If a feature has variance much larger than others, it might dominate the objective function of the estimator. Hence, we applied standard scaling to features such that they have zero mean and unit variance. We then reapplied the same transformation to the testing set[3][4].

To evaluate the performance of the features described in Chapter 3, we experimented with six classification algorithms: Decision Tree, Random Forest, Logistic Regression, K-Nearest Neighbors, SVM, and Naïve Bayes. The classification algorithms were also implemented with scikit-learn. Each classifier can predict a probability for a user-command pair forming a link (positive class) or not (negative class). We used the probability methods offered by scikit-learn to obtain class probabilities for each link. Naïve Bayes and Logistic Regression directly produce probabilities. In K-Nearest Neighbors, the probability of a class for a user-command pair is equal to the fraction of the training examples of the same class in the neighbors of the link. In Decision Tree, the probability of a class for a user-command pair is the fraction of training samples of the same class in a leaf [5]. In Random Forest, the probability of a class is the mean of predicted class of the trees in the forest [6]. Also, SVM can predict probabilities using platt scaling, which is Logistic Regression on SVM's scores [31]. For each user, we sorted their incident edges based on the probabilities and made a top-*N* recommendation list of the corresponding unused commands.

[1] http://scikit-learn.org/stable/modules/grid_search.html
[2] http://scikit-learn.org/stable/
[3] http://scikit-learn.org/stable/modules/preprocessing.html
[4] http://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.StandardScaler.html#sklearn.preprocessing.StandardScaler
[5] http://scikit-learn.org/stable/modules/tree.html#classification
[6] http://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html

5.3.2 CoDis & Variants

In addition to *CoDis*, we further implemented three other variants of our approach to make recommendations. We individually used the *Co-occurrence Score* (*CScore*) and *Discovery Score* (*DScore*) introduced in Sections 4.1 and 4.2, to generate recommendation for users. Moreover, we derived another model *Co+Dis* that combines *CScore* and *DScore* irrespective of elapsed time, in Section 4.4. We implemented all the aforementioned algorithms in Python.

5.3.3 Baselines

We compared all the link prediction algorithms, *CoDis*, *Co+Dis*, *CScore*, and *DScore* against eight baselines, including neighborhood collaborative filtering algorithms that have previously been used for the task of command recommendation [27, 37, 44] and latent factor models that we additionally explore:

- **Non-Personalized Methods**: include the (#1) Most Popular [44] and (#2) Most Widely Used algorithms [27, 33, 37, 44], which have been used previously in the context of command recommendation. The Most Popular algorithm recommends the commands that are the most used by the community. The Most Widely Used algorithm is an extension to the Most Popular that ignores repeated use of a command from a user and recommends the commands that are used by the most number of users in the community.

- **Neighbourhood Models**: include standard (#3) User-based [1] and (#4) Item-based [48] collaborative filtering algorithms, also used in the context of command recommendation in [27, 37]. In the former algorithm, similarities between users are calculated based on their command usage history, and recommendation to users are made based on the commands that similar users use. In the latter algorithm, similarities between commands are calculated based on the users who use them, and recommendation to users are made based on the commands which are similar to the ones they have already used.

- **Collaborative Filtering with Discovery**: include (#7) User-based Discovery and (#8) Item-based Discovery algorithms, which have been proposed in [44] for the task of command recommendation. These methods model users discovery patterns of commands and apply standard neighborhood models on the discovery patterns.

- **Latent Factor Models**: is another primary area for collaborative filtering which explains ratings based on latent factors [25]. To the best of our knowledge, latent factor models have not been previously used for command recommendation. In this work, we used two matrix factorization techniques (#5) Singular Value Decomposition (SVD) [47] and (#6) Adaptive

Gradient (ADAGRAD) [11]. SVD is a technique for dimensionality reduction, where it decomposes a matrix into three lower dimension subcomponents and then reconstructs it for making recommendations. ADAGRAD is a subgradient method to solve regression in matrix factorization, which dynamically adapts the algorithm to the geometry of the data to decompose a matrix into user and item latent factors.

We implemented the Most Popular, Most Widely Used, User-based collaborative filtering, User-based Discovery, and SVD algorithms using Python. For Item-based, Item-based Discovery, and ADAGRAD we used GraphLab[7] [34], a high performance machine learning toolkit.

In the baseline algorithms, similar to [37], we used the tfidf weighting scheme [51] to represent the importance of commands for users. For a user-command pair, this weighting scheme has two components: command frequency and the inverse user frequency. The command frequency represents the importance of a command i to user u, and is defined as the proportion of times command i is used by u, denoted by $cf(u,i)$. The inverse user frequency of command i, denoted by $iuf(i,\mathscr{U})$ describes the discriminative power of the command in the user community, and helps to control for the fact that some commands are generally more common. Altogether, the cfiuf weight for a command is:

$$\begin{aligned} cfiuf(u,i,\mathscr{U}) &= cf(u,i) * iuf(i,\mathscr{U}) \\ &= \frac{\sum_{s=1}^{|\mathscr{S}^u|} n_{s,i}^u}{\sum_{j=1}^{|\mathscr{I}|}\sum_{s=1}^{|\mathscr{S}^u|} n_{s,j}^u} * log\frac{|\mathscr{U}|}{|u \in \mathscr{U} : i \in \mathscr{I}^u|} \end{aligned} \tag{5.1}$$

where the notation is explained in Table 1.1.

For algorithms such as Most Popular, Most Widely Used, User-based collaborative filtering, and SVD, which we implemented, we stored the cfiuf values in a $|\mathscr{U}| * |\mathscr{I}|$ matrix to apply the algorithms.

5.4 Evaluation Metric

A common approach to evaluate recommender systems which use implicit data is through accuracy metrics such as recall and precision [6]. These metrics have been widely used in other domains where explicit preference judgements do not exist [23, 54]. For each user, we make a recommendation list of size N, to predict their next command discovery. We use recall to evaluate performance accuracy:

$$Recall = \frac{\sum_{i=1}^{|\mathscr{U}|} |\mathscr{R}^u \cap \mathscr{Q}_{test}^u|}{|\mathscr{U}|} \tag{5.2}$$

[7] https://dato.com/

where $\mathscr{R}^u$ is defined as the recommendation list for user u, $\mathscr{Q}^u_{test}$ is the testing set for user u, and $|\mathscr{U}|$ is the total number of users in the community. For $k = 1$, $Precision = \frac{Recall}{N}$ and the precision diagram would maintain the same relative ordering as the recall diagram; hence, we do not demonstrate the diagrams for precision separately.

5.5 Parameters

For algorithms that need tuning to avoid over fitting the data, we used a validation set. For the classification algorithms, we first used the k-tail approach described in Section 5.2 to obtain a training set and a testing set. We then used grid search offered by scikit-learn[8] to split the training set into a training set and a validation set. Grid search uses the validation set to avoid over fitting the data on the testing set.

For other algorithms, we first used the k-tail approach described in Section 5.2 to obtain a training set and a testing set. We then used the *k-tail* approach with $k = 1$ on the users training set to further split it into a training set and a validation set. As explained in section 5.1, we conducted experiments on different lengths of all users history. However, to be consistent, we only tuned the parameters on subset (d), which contains the entire command usage history of users who were active during the entire year. We further used the obtained parameters from subset (d) on subsets (a),(b), (c) to show the stability of the *CoDis* algorithm. The tuning lead to the parameters shown in Table 5.1. For algorithms that are dependent on randomization, we used a seed equal to 100. Note that we only report the parameters in which we tuned and do not report the parameters in which we used their default value given by the scikit-learn and GraphLab libraries.

Table 5.1: Parameters used in algorithms.

Algorithm	Parameters	Description
Decision Tree, Random Forest	χ = information gain, σ = best, $\upsilon = 300$	χ is the criterion that measures the quality of a split, σ is the strategy used to choose the split at each node, υ is the number of trees in the forest. All parameters were found using exhaustive search over specified parameter values in subset (d).
Continued on next page		

[8] http://scikit-learn.org/stable/modules/grid_search.html

Table 5.1 Parameters used in algorithms. Continued from previous page

Algorithms	Parameters	Description
Logistic Regression	C=200, Ł = L2,	C is the inverse of the regularization strength, Ł is the norm used for penalization. Both parameters were found using exhaustive search over specified parameter values in subset (d).
K-Nearest Neighbors	$K = 100$, d = Euclidean, ω = inverse of distance	K is the number of neighbors considered for each node, d is the distance metric, ω is the weight function for weighting neighbors. All parameters were found using exhaustive search over specified parameter values in subset (d).
SVM	κ = rbf, $\gamma = \frac{1}{12}$, ϖ = true	κ is the kernel type used, γ is the kernel coefficient, ϖ enables probability estimates in scikit-learn. All parameters were found using exhaustive search over specified parameter values in subset (d).
Naïve Bayes	Γ = Gaussian	Γ is the distribution type. The algorithm performed best with the Gaussian distribution compared to other distributions in all subsets.
CoDis, Co+Dis, *CScore*, *DScore*	$\theta = 100$, $\lambda = 50$	θ is the session chunk threshold in all algorithms, λ is the decay rate in the *CoDis* model. Both parameters were found using exhaustive search over specified parameter values in subset (d).

Continued on next page

Table 5.1 Parameters used in algorithms. Continued from previous page

Algorithms	Parameters	Description
User-based, Item-based, User-based Discovery, Item-based Discovery	$\mu = 32$, $\beta = 40$, $sf =$ cosine similarity	μ is the number of similar users or commands to draw recommendations, β is the base command usage window, which specifies the time in which users know all the commands used upto that point, sf is the function to calculate similarities between users or commands. All parameters were specified in [44]. The algorithms performed best with the cosine similarity function compared to other functions in all subsets.
SVD	$\Delta = 40$	Δ is the dimensionality used to reconstruct the factorized matrix. It was found using exhaustive search over specified parameter values in subset (d).
ADAGRAD	$\delta = 32$	δ is the number of latent factors. It was given by the default settings of GraphLab. Also, the default setting performed better than other settings on average.

The *CoDis* algorithm has two parameters that require tuning: θ the session chunk threshold, and λ the decay rate in the *CoDis* model. The *CoDis* algorithm produced stable results on different values of θ and λ and changed gradually. However, we obtained $\theta = 100s$ and $\lambda = 50$, by tuning the parameters using the entire command usage history of all users.

For ADAGRAD we used the GraphLab library with it's default setting to implement it. Since it is dependent on randomized methods, we repeated each implementation 100 times and reported the average. For the User-based, Item-based, User-based Discovery, and Item-based Discovery methods we used the tune parameters recommended by [37] and [44].

5.6 Experimental Results

Figures 5.1, 5.2, 5.3,and 5.4 demonstrate the performance of the classification algorithms against each other on subsets (a), (b), (c), and (d), respectively. In all figures, recommendation list size, i.e., N, increases from 1 to 20 and k is set to 1. From the classification algorithms, Random Forest performs the best followed by Logistic Regression and K-Nearest Neighbors. Naïve Bayes

performs fourth best because the features are actually correlated and Naïve Bayes assumes they are not. However, the algorithm does not need tuning and is very fast compared to the other algorithms. Surprisingly, SVM with rbf kernel has lower accuracy than the Naïve Bayes algorithm. We conjecture that the low accuracy is the result of inadequate parameter space which we provided for the algorithm, which was itself the result of it's very slow running time. Since SVM with rbf kernel has a very slow running time, we suggest that the other algorithms should be used instead for the command recommendation problem. Decision Tree performs the worst because it over fits the training data. However, Random Forest corrects Decisions Tree's over fitting and performs the best. The accuracy of classifiers such as Random Forest and Logistic Regression on the subsets show that the features described in Section 3.1 are capable of distinguishing between missing and non missing links and provide evidence towards hypothesis 1.

Figures 5.5, 5.6, 5.7, and 5.8 shows the performance of the *CoDis*, *Co+Dis*, *CScore*, *DScore*, all baseline algorithms and the highest performing classification algorithm, i.e., Random Forest, on subsets (a), (b), (c), and (d), respectively. In all figures, recommendation list size, i.e., N, increases from 1 to 20 and k is set to 1. In Figures 5.5, 5.6, 5.7, and 5.8 , we can see that *CoDis* out performs all other algorithms for $N = 3$ to $N = 20$ on all four subsets. The *Co+Dis* performs second best. As the figures demonstrate, *Codis* always has a higher recall than *Co+Dis*. Therefore, considering the elapsed time of users as an indicator of the type of recommendation they require is effective and leads to better accuracy.

For $N = 20$, Random Forest performs third best in all subset except for (a). This is probably due to the lack of positive examples in subset (a). In subset (d), from among the baseline algorithms, User-based Discovery performs the best for $N = 1$ to $N = 15$,. However, for $N = 15$ to $N = 20$ ADAGRAD performs the best. The Most popular performs the worst in (b), (c), and (d). However, in (a), it performs as good as ADAGRAD.

In subset (d), User-based Discovery performs the best for $N = 1$ to $N = 2$. For User-based discovery and Item-based discovery algorithms, Murphy-Hill and colleagues [44] used a base command usage window of size 40 sessions, to discard false positive discoveries. As explained in Section 2.1.2, they assumed a command discovery occurs only after the base command usage window and only if it is used more than once. These requirements reduced (d) to 3724 users and 614 commands for the User-based Discovery and Item-based Discovery algorithms (Figure 5.8). In the other subsets, the number of users is reduced to less than 2600 users and the number of commands is also dropped. As a result, we chose not to report the recall of User-based Discovery and Item-based Discovery in subsets (a), (b), and (c) as we thought it is not a fair comparison to other algorithms.

Based on the users entire command usage history (subset d), for the *DScore* and *CScore* models, we computed for which common users they made a correct recommendation. Figure 5.9

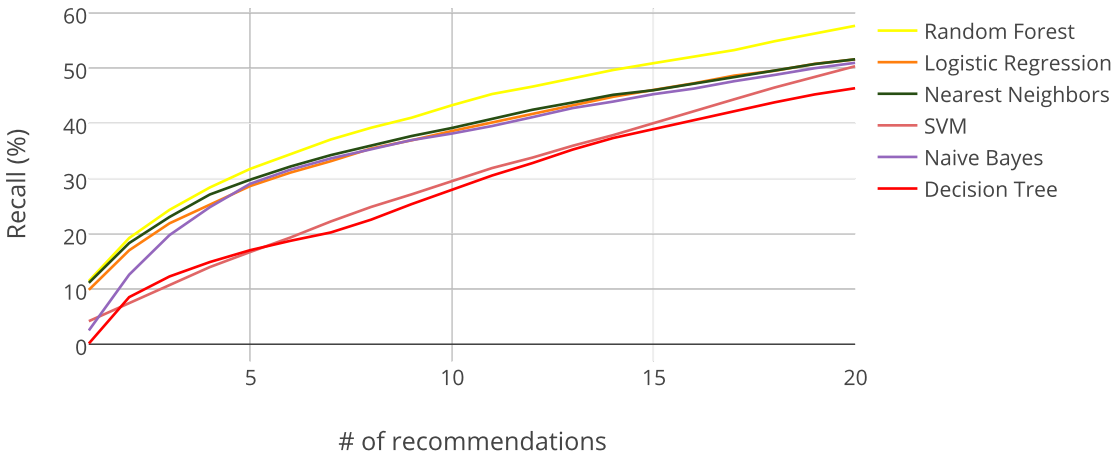


Figure 5.1: Recall of link prediction classifiers in subset (a). Dataset consists of 4178 users and 480 commands.

demonstrates their Jaccard similarity for the top-N recommendation task, where N changes from 1 to 20. The two sets of user have less than 35% overlap for all N. This indicates that the two algorithms produce different recommendations and are able to make correct recommendations for different sets of users (Figure 5.9).

Moreover, we analysed the elapsed time of the users of subset (d) for which there was a correct recommendation, from the *CoDis*, *Co+Dis*, *CScore*, and *DScore* algorithms. Figure 5.10 demonstrates their average elapsed time for the top-N recommendation task, where N changes from 1 to 20. In Figure 5.10, we can see that the average elapsed time for the *CScore* model is very low compared to *DScore* model. This means that the *CScore* model is able to produce correct recommendations for users with small elapsed time, whereas the *Discovery* model is able to make correct recommendations for users with large elapsed time. The *CoDis* algorithm which takes into account elapsed time directly in the model, has higher average elapsed time compared to *Co+Dis*. However, both *CoDis* and *Co+Dis* converge to the average elapsed time of the entire users, which is 43.40 hours.

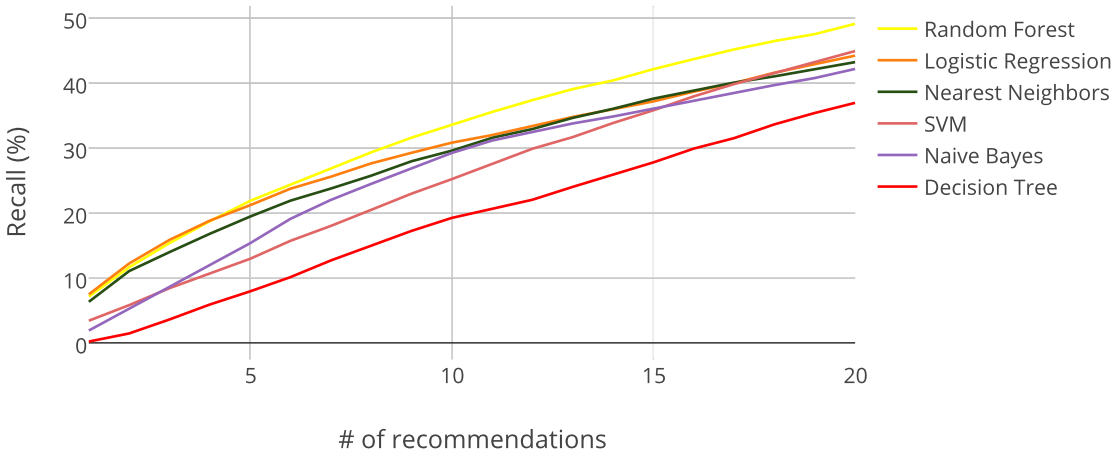


Figure 5.2: Recall of link prediction classifiers in subset (b). Dataset consists of 4292 users and 638 commands.

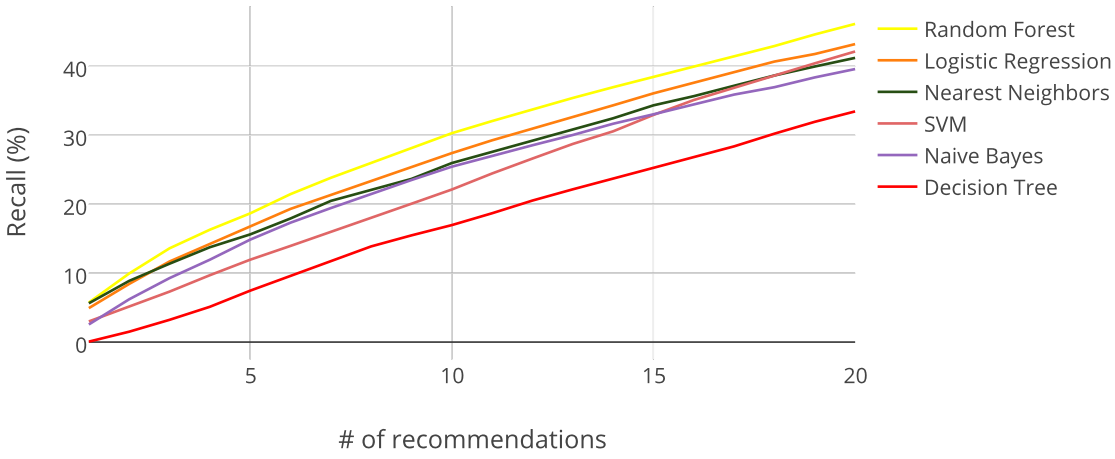


Figure 5.3: Recall of link prediction classifiers in subset (c). Dataset consists of 4301 users and 676 commands.

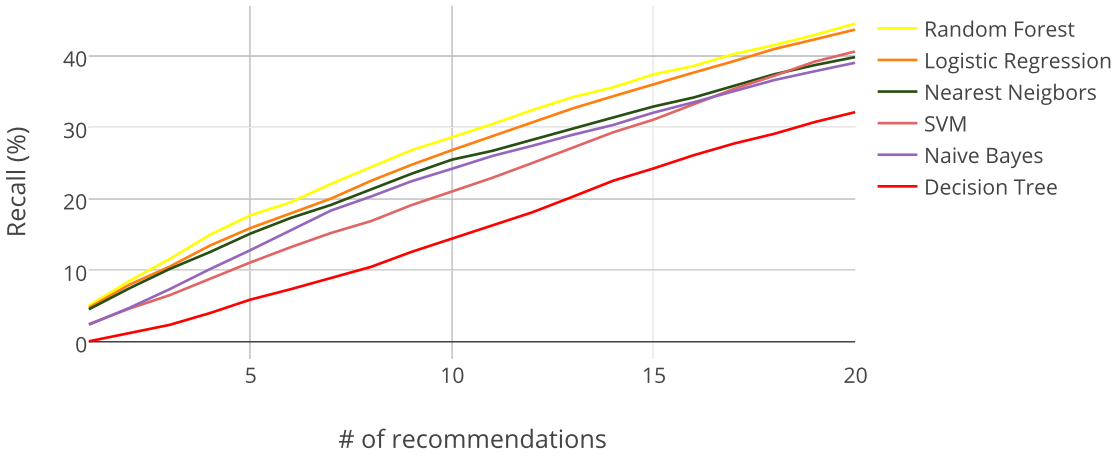


Figure 5.4: Recall of link prediction classifiers in subset (d). Dataset consists of 4305 users and 700 commands. Item-based Discovery uses 3724 users and 614 commands.

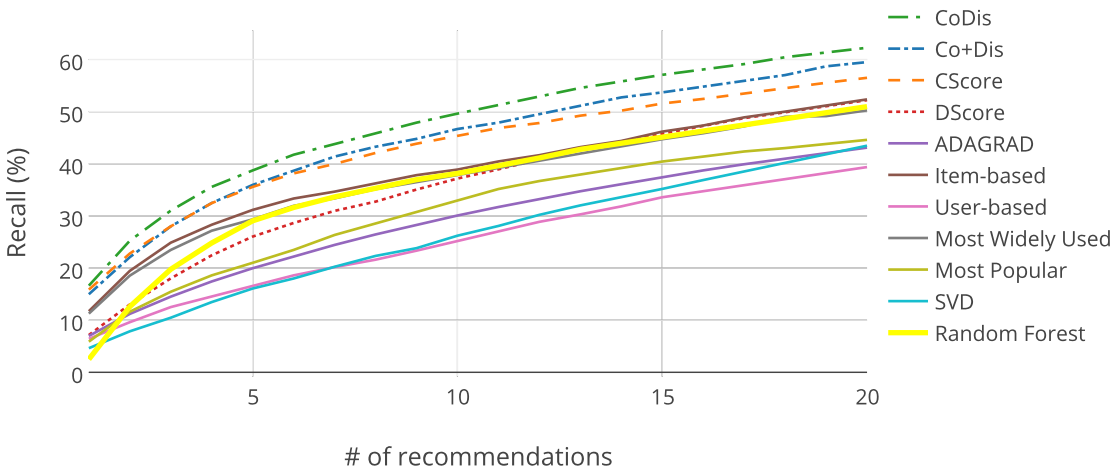


Figure 5.5: Recall of *CoDis*, *Co + Dis*, *CScore*, and *DScore* algorithms against baselines in subset (a). Dataset consists of 4178 users and 480 commands.

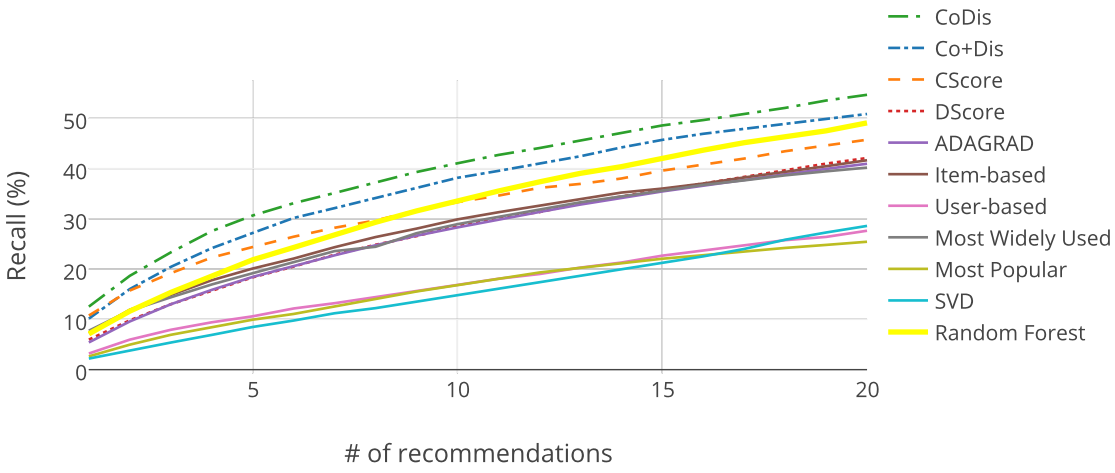


Figure 5.6: Recall of *CoDis*, *Co + Dis*, *CScore*, and *DScore* algorithms against baselines in subset (b). Dataset consists of 4292 users and 638 commands.

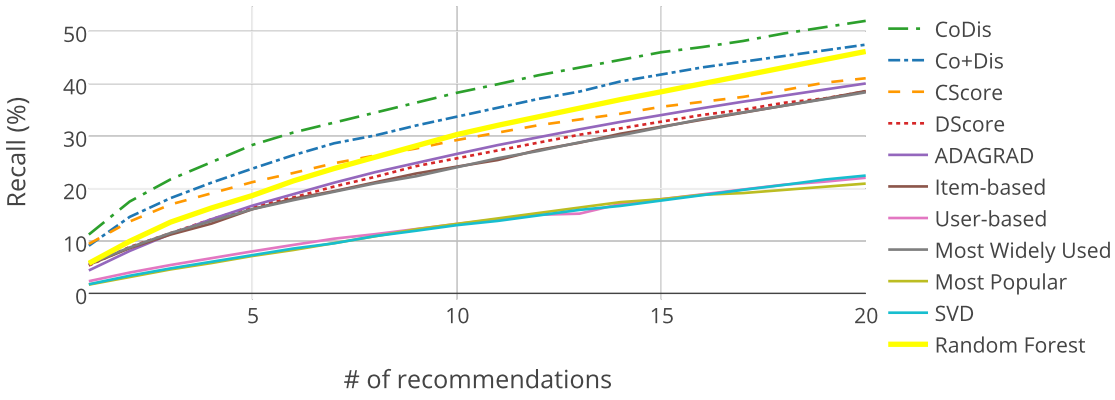


Figure 5.7: Recall of *CoDis*, *Co + Dis*, *CScore*, and *DScore* algorithms against baselines in subset (c). Dataset consists of 4301 users and 676 commands.

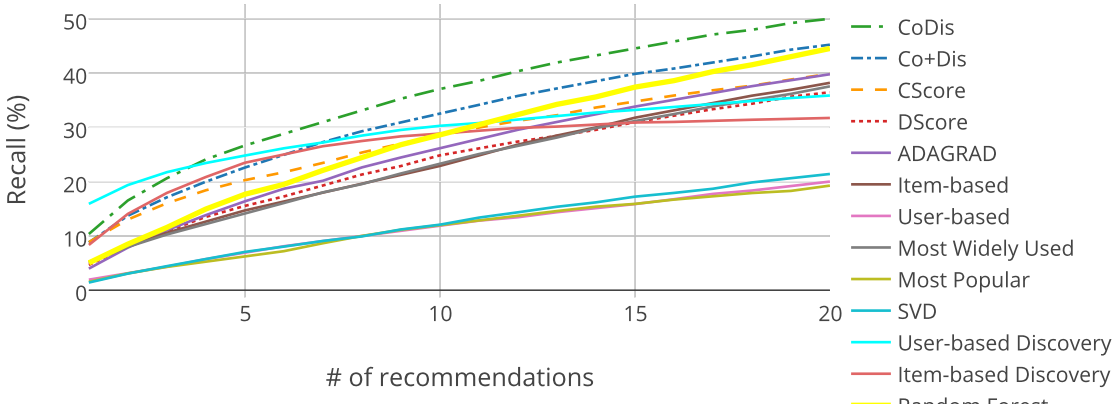


Figure 5.8: Recall of *CoDis*, *Co + Dis*, *CScore*, and *DScore* algorithms against baselines in subset (d). Dataset consists of 4305 users and 700 commands. User-based Discovery and Item-based Discovery use a dataset consisting of 3724 users and 614 commands.

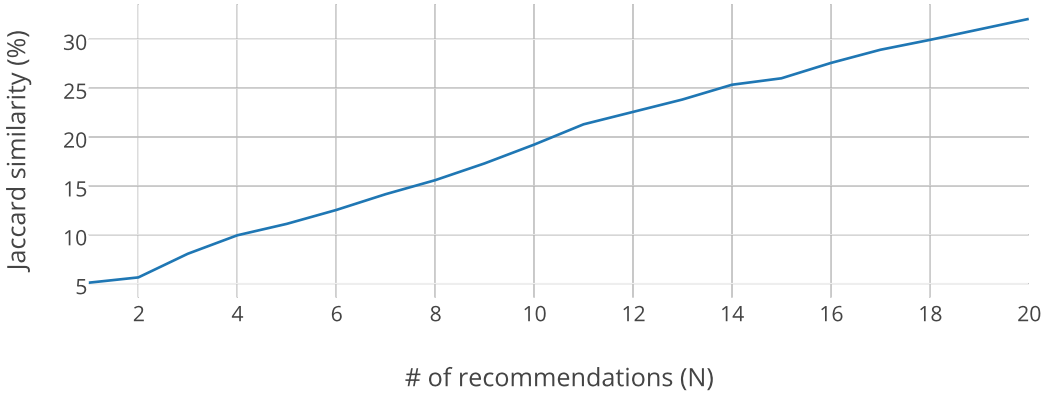


Figure 5.9: Jaccard similarity of users that the *CScore* and *DScore* algorithms were able to make correct recommendation for, in subset (d).

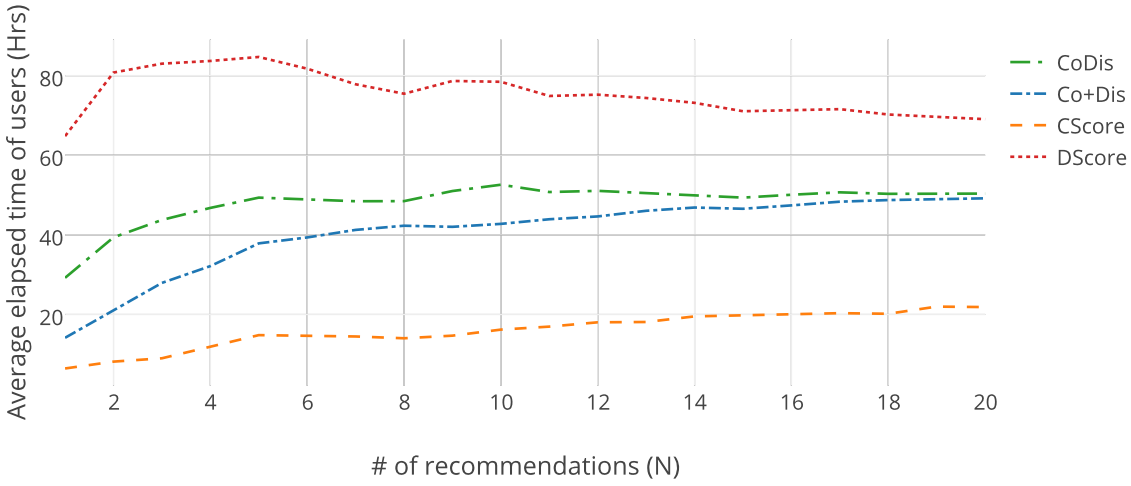


Figure 5.10: Average elapsed time of users that the *CoDis*, *Co+Dis*, *CScore*, and *DScore* algorithms were able to make correct recommendation for, in subset(d)

5.7 Link Prediction Feature Comparison

One of our objectives was to compare the 12 features described in Section 3.1, to judge their relative strength in a link prediction task. Figures 5.11 and 5.12 provide a comparison of the 12 features for subset (d). They show the features rank of importance in information gain as obtained by the Decision Tree and Random Forest algorithms, respectively. As illustrated in Figures 5.11 and 5.12, Command Common Users Jaccards Coefficient has the highest rank in the Eclipse dataset. In Figure 5.11, all other features have low information gain, whereas in figure 5.12, all features have relatively high information gain. Because Random Forest corrects Decision Tree's overfitting, its ranking of attributes are more likely to be correct. Therefore, the combination of features used in the link prediction problem act as good discriminators for missing and non-missing links when they are combined compared to when they are used solely.

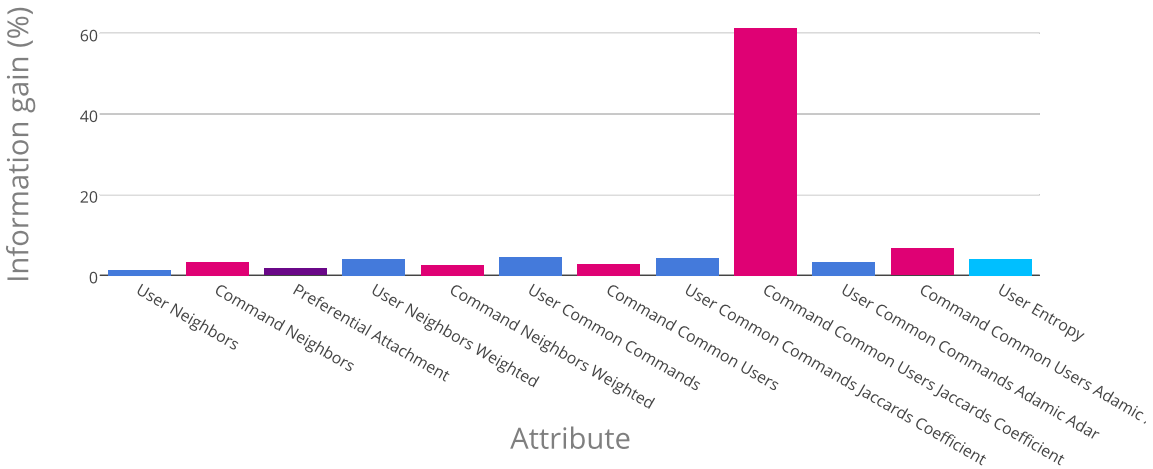


Figure 5.11: Information gain of features based on the Decision Tree classifier, in subset (d).

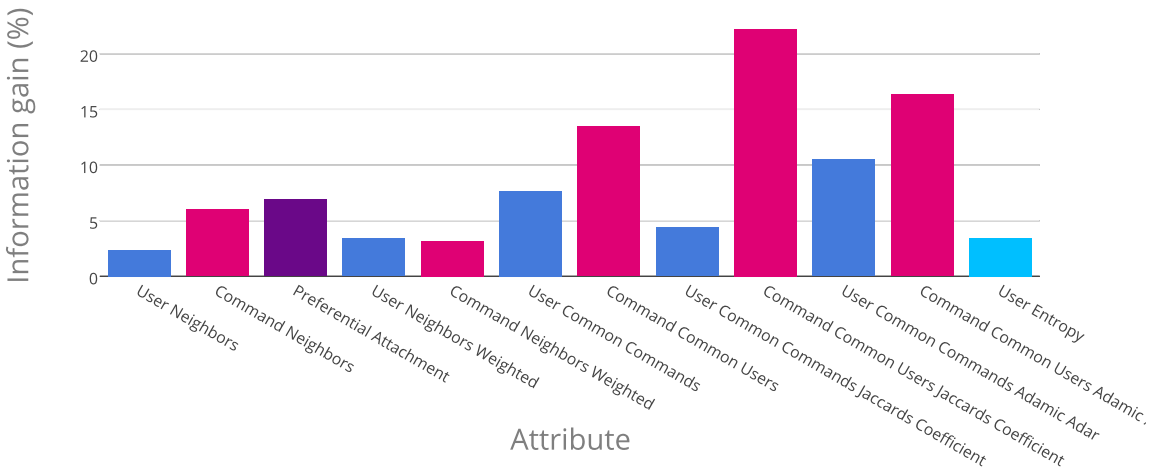


Figure 5.12: Information gain of features based on the Random Forest classifier, in subset (d).

5.8 Summary of Results

Experiments on the Eclipse dataset show that from among the link prediction algorithms, Random Forest performs the best. Compared to the matrix factorization technique with ADAGRAD solver [11], that has the best accuracy amongst the baselines, Random Forest obtained an improvement of 4.74% in terms of the recall performance measure for a top-*N* recommendation task, where $N = 20$. Random Forest suggests that the features provided all have relatively high information gain, but the Command Common Users Jaccards Coefficient has the highest.

The *CoDis* algorithm achieved significant performance improvements, in terms of recall, over all link prediction and baseline algorithms. Compared to the matrix factorization technique with ADAGRAD solver, *CoDis* obtained an improvement of 10.22% in terms of the recall performance measure for a top-*N* recommendation task, where $N = 20$. Based on the *CoDis* algorithm, the results suggest that taking into account the elapsed time between users last activity time and time of recommendation could be useful for producing more accurate recommendations.

Chapter 6

Discussion

In this chapter, we discuss some disadvantages and advantages of our approaches. A salient limitation of the link prediction approach is the fact that not all features were explored. Other similarity features in the user-command network such as Hitting time [29], which is the expected number of steps required for a random walk to start at a node and reach the other node, can be investigated to measure the similarity between a user node and a command node. Also, other user features such as skewness [10], which measures whether a user has used many commands with a skewed distribution, can be explored.

A common limitation of the link prediction approach and the *CoDis* approach is the fact that we considered a user-command pair in which the user has used the command at least once a positive example, and a negative example otherwise. However, a user may use a command several times and later realise they don't like it, or a user might actually find a command useful but never use it due to unawareness of its existence. Hence, categorizing user-command pairs as positive or negative examples, should be handled more carefully. For example, Murphy-Hill and colleagues [44] use concepts such as k-tail multi-use, which means a command should be used multiple times by a user to be considered as a positive example, to categorize user-command pairs. Similar approaches can be applied to both the link prediction algorithms and the *CoDis* algorithm to improve quality of recommendation for users.

One drawback of the *CoDis* algorithm is identifying co-occurrence and discovery patterns in datasets. Since commands can be executed in any order, two co-occurring commands may be executed far apart in time, even in different sessions. Also, two commands that are discovered in the same session might actually form a discovery pattern. Distinguishing between these situations is even harder in datasets which do not provide explicit user sessions. For a co-occurrence pattern to be missed, the users should use the co-occurring commands far apart most of the times. If the command has been used very rare by the users and also far apart then it is likely to be missed as

a co-occurring pattern. Also, if two commands of a discovery pattern are discovered in the same session or are discovered not consecutively, then it is likely to be missed as a discovery pattern.

Moreover, we assumed when a user first uses a command, then the user has discovered it. Since we might not have a user's entire command usage history, a command discovery might not actually be a discovery. In other words, when a user issues a command for the first time it does not necessarily mean that the command was just learned. The user may have known the command, despite not using it. There exists different techniques to rule out such false-positive discoveries. For example, Murphy-Hill and colleagues [44] consider a command usage as a command discovery only after a base time, which represents a period of time for which they assume the developers know the command.

We further explored other definitions for discovery patterns, but they led to lower accuracy. For example, an alternative for Definition 2 in which we explored is:

Definition 4. [Command Discovery Pattern Alternative] We assume users discover a command when they first execute the command. A command discovery pattern (i, j) happens when a user first discovers command i in a session and discovers command j consecutively in a later session.□

According to this definition, if a user discovers commands i and j in their first session, reuses j in their second session, and reuses i and discovers k in their third session. Then both $df_{i,k}$ and $df_{j,k}$ increase by one, because i and j were discovered in the first session and k was consecutively discovered in the third session.

Another drawback of the *CoDis* algorithm is that Hypothesis 4 assumes there is a change in task when elapsed time is large and that not might be true. A user might require recommendation related to their current task even though they have a large elapsed time.

Overall, based on the *CoDis* algorithm, the results suggest that taking into account the elapsed time between users last activity time and time of recommendation could be useful for producing more accurate recommendations. However, accuracy is not the only important factor for evaluating recommender systems. Another key challenge is *explainability* [24]. For each recommendation to a user, we can show the co-occurrence and discovery patterns which had the highest impact. Specifically, in the context of recommending software tools, learning from peers is very effective but happens very rare due to limitations [43]. Hence, recommending prevalent patterns in the user community can improve users awareness.

6.1 Time Complexity

In addition to accuracy and explainability, another good reason for deploying the *CoDis* algorithm is computational efficiency, which is a key aspect in recommender systems [24]. The link prediction and baseline algorithms may have different time complexities based on their implementation. In the

following, we give an approximation of all algorithms' time complexity based on the solvers that scikit-learn and GraphLab use. For the algorithms that we implemented from scratch, we present a time analysis based on our implementation.

Let $\mathscr{U}$ be the users set, $\mathscr{I}$ be the set of commands in which the users have used, $\mathscr{Q}$ be the entire command usage history of the users. For the classification algorithms, let $\mathscr{A}_{train}$ be the instances in the training set, $\mathscr{A}_{test}$ be the instances in the testing set, and d be the number of attributes. Note that $|\mathscr{A}_{train}| + |\mathscr{A}_{test}| = |\mathscr{U}| * |\mathscr{I}|$, since each positive instance is weighted by number of times the user uses the command.

All the algorithms have time complexity at least $\Omega(|\mathscr{Q}|)$ to scan the input. However, we exclude it from their runtime given below. Also, we exclude the time of ranking recommendations for users.

For link prediction, creating a sophisticated feature set might require a noticeable amount of time. Some features have trivial time complexity, whereas some have salient time complexity. For example, for a single user node, extracting the User Neighbors feature is $O(|\mathscr{I}|)$. Whereas extracting User Common Commands for a user-command pair is $O(|\mathscr{U}||\mathscr{I}|)$. Also, the classification algorithms might have noticeable time complexity. Decision Tree has time complexity $O(d|\mathscr{A}_{train}|log|\mathscr{A}_{train}|)$ [57]. Random Forest on a single core with υ trees has time complexity $O(\upsilon d|\mathscr{A}_{train}|log|\mathscr{A}_{train}|)$. For SVD, scikit-learn's implementation is based on libsvm and has time complexity between $O(d|\mathscr{A}_{train}|^2)$ and $O(d|\mathscr{A}_{train}|^3)$ [1] [5] to train data. For Logistic Regression, we used the liblinear solver of scikit-learn. The liblinear solver uses a coordinate descent algorithm based on liblinear [12], where each iteration is $O(d|\mathscr{A}_{train}|)$ [59]. For K-nearest Neighbors, training isn't required, but classifying each instance requires $O(d|\mathscr{A}_{train}|)$ using brute force search [2]. scikit-learn provides several approximate algorithms to reduce the time complexity at the cost of accuracy. Training with Naïve Bayes has optimal time complexity $O(d|\mathscr{A}_{train}|)$ [36], since it is linear in the time it takes to scan all instances.

The Most Popular and Most Widely Used algorithms require $O(|\mathscr{U}||\mathscr{I}|)$ time, since for each command we need to acquire all the users who use it. Item-based collaborative filtering has time complexity $O(|\mathscr{U}||\mathscr{I}|^2)$ for training and has time complexity $O(|\mathscr{U}||\mathscr{I}|)$ for generating predictions for a single user [14, 32]. User-based collaborative filtering doesn't require training, however to make predictions for a single user, it requires $O(|\mathscr{U}||\mathscr{I}|)$ [14] time. In Item-based Discovery, items are command discovery patterns and the number of command discovery patterns is $O(|\mathscr{I}|^2)$. Hence, Item-based Discovery has runtime $O(|\mathscr{U}||\mathscr{I}|^4)$ for training and $O(\mathscr{U}|\mathscr{I}|^2)$ to make predictions for a single user. In User-based Discovery, items are also command discovery patterns. Therefore, to make predictions for a single user, $O(|\mathscr{U}||\mathscr{I}|^2)$ time is required. The fastest algorithms for SVD are proportional to $O(|\mathscr{U}|^2|\mathscr{I}| + |\mathscr{I}|^3)$ [18], which is equal to $O(|\mathscr{U}|^2|\mathscr{I}|)$ for

[1] http://scikit-learn.org/stable/modules/svm.html#complexity

[2] http://scikit-learn.org/stable/modules/neighbors.html#brute-force

$|\mathscr{I}| \leq |\mathscr{U}|$. Each iteration of ADAGRAD is $O(QN)$, where N is the number of sub-functions used in the algorithm and Q is the cost of computing the value and gradient for a single sub-function [50].

In the *CoDis* algorithm, while the users command usage history is being scanned, the **DF** and **CF** matrices can be generated. The **CP** and **DP** matrices can be obtained in $O(|\mathscr{I}|^2)$, since we need to compute the sum of each row once and then normalize each cell by the sum of it's corresponding row. To compute *CScore* for a single user, all the commands in their last session and their co-occurring commands need to be investigated. Hence, *CScore* is $O(|\mathscr{I}|^2)$ in worst case. To compute *DScore* for a single user, all the commands in which the user has discovered and the commands that have been discovered after them need to be investigated. Therefore, *DScore* also has time complexity $O(|\mathscr{I}|^2)$. For each user and command, obtaining *CoDis* from their *CScore* and *DScore* requires $O(1)$ time. Hence, to make prediction for a single user, *CoDis* requires $O(\mathscr{I}^2)$ time. Similarly, to make predictions for a single user, *Co+Dis* has time complexity $O(\mathscr{I}^2)$. Since the **CP** and **DP** matrices can be computed offline and in parallel, then computing the scores for a user can be done online when a user asks for recommendation. Hence, the personalized recommendations can be produced in a timely manner when a user asks for recommendation. Therefore, the *CoDis* algorithm can be easily scaled to millions of users and commands.

Chapter 7

Conclusion and Future Work

In this work, we addressed the command recommendation problem. We proposed a link prediction approach and a novel algorithm called *CoDis* that can be deployed to teach users in a developer community what command to learn next.

For the link prediction approach, we obtained a user-command bipartite network from command usage history of users and extracted topological features from the network for user-command pairs. We used several classification algorithms to infer for each user, which commands they are likely to form an link with and generated recommendations based on the predicted links.

Our proposed algorithm, *CoDis*, generates personalized recommendations for a user by analysing the user's past command usage history, co-occurrence and discovery patterns of commands within the entire community, and also the elapsed time between the user's last activity and the time of recommendation. The algorithm first extracts co-occurrence and discovery patterns from the command usage history of the entire user community. Then for each user, computes a discovery and co-occurrence score based on their own command usage history. Finally, for each user, it weighs their co-occurrence score and discovery score according to their own elapsed time. The *CoDis* considers the users elapsed time as an indicator of their need. In particular, *CoDis* assumes if a user's elapsed time is relatively small, they are more likely to be working on their most recent tasks, and if the elapsed time is relatively large, the user is more likely to have begun a new task. Based on this intuition, if the users elapsed time is relatively low, then *CoDis* generates recommendations that take into account the co-occurrence score more than the discovery score and if the users elapsed time is relatively high, then it generates recommendations that take into account the discovery score more than the co-occurrence score.

We evaluated our approaches on data submitted by anonymous users of the Eclipse IDE, in terms of recall. We compared the accuracy of the classifiers and the *CoDis* algorithm with standard algorithms used for command recommendation and matrix factorization techniques that are

known to perform well in other domains. The *CoDis* algorithm performed better than all link prediction and baseline algorithms and the Random Forest classifier performed better than all other classification algorithms. *CoDis* and *Random Forest* achieved an increase of 10.22% and 4.74% in recall, compared to the matrix factorization technique with ADAGRAD solver, the best performing baseline, for a top-N recommendation task, where $N = 20$. Based on the *CoDis* algorithm, the results suggest that taking into account the elapsed time between users last activity time and time of recommendation could be useful for producing more accurate recommendations.

Building on our results, there are several promising areas for future work. A promising direction for the *CoDis* algorithm, is to investigate other models for taking advantage of the discovery and co-occurrence scores. In *CoDis*, we used a linear combination of co-occurrence and discovery scores weighted by elapsed time to obtain final scores. We used a tuning parameter λ to specify whether users prefer recommendation based on co-occurrence patterns or discovery patterns. Alternative models, such as exponential functions with a decay rate [9] can be explored for combining the co-occurrence and discovery scores.

Moreover, other patterns can be taken into account for estimating the co-occurrence and discovery scores. Currently, for both types of patterns, we have considered 2-grams. Patterns of larger sizes, e.g., 3-grams or 4-grams, should be investigated. Further, in the *CoDis* algorithm, to compute *CScore* we used a weighting function to weigh sessions importance, where the last session weighed one and previous sessions weighed zero. Future work should investigate whether other weighting functions which give more weight to the user's previous sessions, perform better or not.

Although our analysis and experiments focus on recommending IDE commands, we believe that our model can be extended in applications where learning takes place, such as other feature-rich applications and application programming interfaces. Our model can be explored for deployment in these tools to teach users how to navigate through them. Also, it is worthy to conduct experimental studies in non-technical environments where learning takes place. For example, for teaching language related skills to users.

Bibliography

[1] G. Adomavicius and A. Tuzhilin. Toward the next generation of recommender systems: A survey of the state-of-the-art and possible extensions. *Knowledge and Data Engineering, IEEE Transactions on*, 17(6):734–749, 2005. → pages 26

[2] M. Al Hasan, V. Chaoji, S. Salem, and M. Zaki. Link prediction using supervised learning. In *SDM06: Workshop on Link Analysis, Counter-terrorism and Security*, 2006. → pages 3, 11

[3] D. M. Blei, A. Y. Ng, and M. I. Jordan. Latent dirichlet allocation. *the Journal of machine Learning research*, 3:993–1022, 2003. → pages 4

[4] L. Breiman. Random forests. *Machine learning*, 45(1):5–32, 2001. → pages 15

[5] C.-C. Chang and C.-J. Lin. Libsvm: A library for support vector machines. *ACM Transactions on Intelligent Systems and Technology (TIST)*, 2(3):27, 2011. → pages 43

[6] P. Cremonesi, Y. Koren, and R. Turrin. Performance of recommender algorithms on top-n recommendation tasks. In *Proceedings of the fourth ACM conference on Recommender systems*, pages 39–46. ACM, 2010. → pages 27

[7] P. Cunningham and S. J. Delany. k-nearest neighbour classifiers. *Mult Classif Syst*, pages 1–17, 2007. → pages 16

[8] M. Damashek et al. Gauging similarity with n-grams: Language-independent categorization of text. *Science*, 267(5199):843–848, 1995. → pages 4, 11

[9] Y. Ding and X. Li. Time weight collaborative filtering. In *Proceedings of the 14th ACM international conference on Information and knowledge management*, pages 485–492. ACM, 2005. → pages 46

[10] D. P. Doane and L. E. Seward. Measuring skewness: a forgotten statistic. *Journal of Statistics Education*, 19(2):1–18, 2011. → pages 41

[11] J. Duchi, E. Hazan, and Y. Singer. Adaptive subgradient methods for online learning and stochastic optimization. *The Journal of Machine Learning Research*, 12:2121–2159, 2011. → pages 5, 27, 40

[12] R.-E. Fan, K.-W. Chang, C.-J. Hsieh, X.-R. Wang, and C.-J. Lin. Liblinear: A library for large linear classification. *The Journal of Machine Learning Research*, 9:1871–1874, 2008. → pages 43

[13] L. Findlater and J. McGrenere. A comparison of static, adaptive, and adaptable menus. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 89–96. ACM, 2004. → pages 9

[14] V. Formoso, F. Cacheda, and V. Carneiro. Algorithms for efficient collaborative filtering. In *Efficiency Issues in Information Retrieval Workshop*, page 17, 2008. → pages 43

[15] A. Fourney, R. Mann, and M. Terry. Query-feature graphs: bridging user vocabulary and system functionality. In *Proceedings of the 24th annual ACM symposium on User interface software and technology*, pages 207–216. ACM, 2011. → pages 8

[16] K. Z. Gajos, M. Czerwinski, D. S. Tan, and D. S. Weld. Exploring the design space for adaptive graphical user interfaces. In *Proceedings of the working conference on Advanced visual interfaces*, pages 201–208. ACM, 2006. → pages 9

[17] F. Garcin, K. Zhou, B. Faltings, and V. Schickel. Personalized news recommendation based on collaborative filtering. In *Proceedings of the The 2012 IEEE/WIC/ACM International Joint Conferences on Web Intelligence and Intelligent Agent Technology-Volume 01*, pages 437–441. IEEE Computer Society, 2012. → pages 11

[18] G. H. Golub and C. F. Van Loan. *Matrix computations*, volume 3. JHU Press, 2012. → pages 44

[19] J. Han, H. Cheng, D. Xin, and X. Yan. Frequent pattern mining: current status and future directions. *Data Mining and Knowledge Discovery*, 15(1):55–86, 2007. → pages 4

[20] E. Horvitz, J. Breese, D. Heckerman, D. Hovel, and K. Rommelse. The lumiere project: Bayesian user modeling for inferring the goals and needs of software users. In *Proceedings of the Fourteenth conference on Uncertainty in artificial intelligence*, pages 256–265. Morgan Kaufmann Publishers Inc., 1998. → pages 9

[21] Z. Huang, X. Li, and H. Chen. Link prediction approach to collaborative filtering. In *Proceedings of the 5th ACM/IEEE-CS joint conference on Digital libraries*, pages 141–142. ACM, 2005. → pages 3, 11, 13, 14, 15

[22] M. A. A. Khan, V. Dziubak, and A. Bunt. Exploring personalized command recommendations based on information found in web documentation. In *Proceedings of the 20th International Conference on Intelligent User Interfaces*, pages 225–235. ACM, 2015. → pages 8

[23] I. Konstas, V. Stathopoulos, and J. M. Jose. On social networks and collaborative recommendation. In *Proceedings of the 32nd international ACM SIGIR conference on Research and development in information retrieval*, pages 195–202. ACM, 2009. → pages 27

[24] Y. Koren. Tutorial on recent progress in collaborative filtering. *RecSys*, 8:333–334, 2008. → pages 42

[25] Y. Koren, R. Bell, and C. Volinsky. Matrix factorization techniques for recommender systems. *Computer*, 42(8):30–37, 2009. → pages 26

[26] B. Lafreniere, A. Bunt, M. Lount, and M. A. Terry. Understanding the roles and uses of web tutorials. In *ICWSM*, 2013. → pages 7

[27] W. Li, J. Matejka, T. Grossman, J. A. Konstan, and G. Fitzmaurice. Design and evaluation of a command recommendation system for software applications. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 18(2):6, 2011. → pages 1, 3, 10, 24, 26

[28] X. Li and H. Chen. Recommendation as link prediction in bipartite graphs: A graph kernel-based machine learning approach. *Decision Support Systems*, 54(2):880–890, 2013. → pages 3

[29] D. Liben-Nowell and J. Kleinberg. The link-prediction problem for social networks. *Journal of the American society for information science and technology*, 58(7):1019–1031, 2007. → pages 3, 10, 11, 41

[30] R. N. Lichtenwalter, J. T. Lussier, and N. V. Chawla. New perspectives and methods in link prediction. In *Proceedings of the 16th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 243–252. ACM, 2010. → pages 11

[31] H.-T. Lin, C.-J. Lin, and R. C. Weng. A note on platts probabilistic outputs for support vector machines. *Machine learning*, 68(3):267–276, 2007. → pages 25

[32] G. Linden, B. Smith, and J. York. Amazon. com recommendations: Item-to-item collaborative filtering. *Internet Computing, IEEE*, 7(1):76–80, 2003. → pages 43

[33] F. Linton, D. Joy, H.-P. Schaefer, and A. Charron. Owl: A recommender system for organization-wide learning. *Educational Technology & Society*, 3(1):62–76, 2000. → pages 9, 26

[34] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein. Distributed graphlab: a framework for machine learning and data mining in the cloud. *Proceedings of the VLDB Endowment*, 5(8):716–727, 2012. → pages 27

[35] K. Lubick, T. Barik, and E. Murphy-Hill. Can social screencasting help developers learn new tools? 2015. → pages 8

[36] C. D. Manning, P. Raghavan, H. Schütze, et al. *Introduction to information retrieval*, volume 1. Cambridge university press Cambridge, 2008. → pages 43

[37] J. Matejka, W. Li, T. Grossman, and G. Fitzmaurice. Communitycommands: Command recommendations for software applications. In *Proceedings of the 22Nd Annual ACM Symposium on User Interface Software and Technology*, UIST '09, pages 193–202, New

York, NY, USA, 2009. ACM. ISBN 978-1-60558-745-5. doi:10.1145/1622176.1622214. URL http://doi.acm.org/10.1145/1622176.1622214. → pages 1, 3, 10, 24, 26, 27, 30

[38] J. Matejka, T. Grossman, and G. Fitzmaurice. Ip-qat: in-product questions, answers, & tips. In *Proceedings of the 24th annual ACM symposium on User interface software and technology*, pages 175–184. ACM, 2011. → pages 8

[39] R. L. Mercer et al. Class-based n-gram models of natural language, 1990. → pages 4, 11

[40] D. Michie, D. J. Spiegelhalter, and C. C. Taylor. Machine learning, neural and statistical classification. 1994. → pages 11, 15, 16

[41] K. P. Murphy. *Machine learning: a probabilistic perspective*. MIT press, 2012. → pages 16

[42] E. Murphy-Hill. Continuous social screencasting to facilitate software tool discovery. In *Proceedings of the 34th International Conference on Software Engineering*, pages 1317–1320. IEEE Press, 2012. → pages 8

[43] E. Murphy-Hill and G. C. Murphy. Peer interaction effectively, yet infrequently, enables programmers to discover new tools. In *Proceedings of the ACM 2011 conference on Computer supported cooperative work*, pages 405–414. ACM, 2011. → pages 8, 42

[44] E. Murphy-Hill, R. Jiresal, and G. C. Murphy. Improving software developers' fluency by recommending development environment commands. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, page 42. ACM, 2012. → pages 4, 10, 20, 23, 24, 26, 30, 31, 41, 42

[45] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011. → pages 25

[46] M. A. Saied, O. Benomar, H. Abdeen, and H. Sahraoui. Mining multi-level api usage patterns. In *Software Analysis, Evolution and Reengineering (SANER), 2015 IEEE 22nd International Conference on*, pages 23–32. IEEE, 2015. → pages 4

[47] B. Sarwar, G. Karypis, J. Konstan, and J. Riedl. Application of dimensionality reduction in recommender system-a case study. Technical report, DTIC Document, 2000. → pages 26

[48] B. Sarwar, G. Karypis, J. Konstan, and J. Riedl. Item-based collaborative filtering recommendation algorithms. In *Proceedings of the 10th international conference on World Wide Web*, pages 285–295. ACM, 2001. → pages 26

[49] T. Segaran. *Programming collective intelligence: building smart web 2.0 applications*. " O'Reilly Media, Inc.", 2007. → pages 15

[50] J. Sohl-Dickstein, B. Poole, and S. Ganguli. Fast large-scale optimization by unifying stochastic gradient and quasi-newton methods. *arXiv preprint arXiv:1311.2115*, 2013. → pages 44

[51] K. Sparck Jones. A statistical interpretation of term specificity and its application in retrieval. *Journal of documentation*, 28(1):11–21, 1972. → pages 10, 27

[52] R. Srikant and R. Agrawal. *Mining sequential patterns: Generalizations and performance improvements*. Springer, 1996. → pages 4

[53] Y. Sun, R. Barber, M. Gupta, C. C. Aggarwal, and J. Han. Co-author relationship prediction in heterogeneous bibliographic networks. In *Advances in Social Networks Analysis and Mining (ASONAM), 2011 International Conference on*, pages 121–128. IEEE, 2011. → pages 3, 10

[54] M. Trevisiol, L. M. Aiello, R. Schifanella, and A. Jaimes. Cold-start news recommendation with domain-dependent browse graph. In *Proceedings of the ACM Recommender System conference, RecSys*, volume 14, 2014. → pages 11, 27

[55] M. B. Twidale. Over the shoulder learning: supporting brief informal learning. *Computer Supported Cooperative Work*, 14(6):505–547, 2005. → pages 8

[56] P. Viriyakattiyaporn and G. C. Murphy. Improving program navigation with an active help system. In *Proceedings of the 2010 Conference of the Center for Advanced Studies on Collaborative Research*, pages 27–41. IBM Corp., 2010. → pages 4, 9

[57] I. H. Witten and E. Frank. *Data Mining: Practical machine learning tools and techniques*. Morgan Kaufmann, 2005. → pages 43

[58] H. Yin, B. Cui, J. Li, J. Yao, and C. Chen. Challenging the long tail recommendation. *Proceedings of the VLDB Endowment*, 5(9):896–907, 2012. → pages 15

[59] H.-F. Yu, F.-L. Huang, and C.-J. Lin. Dual coordinate descent methods for logistic regression and maximum entropy models. *Machine Learning*, 85(1-2):41–75, 2011. → pages 43

[60] E. Zheleva, L. Getoor, J. Golbeck, and U. Kuter. Using friendship ties and family circles for link prediction. In *Advances in Social Network Mining and Analysis*, pages 97–113. Springer, 2010. → pages 10