Automatic Identification and Description of Software Developers Tasks

by

Christopher Satterfield

B.Sc, University of British Columbia, 2018

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)

April 2020

© Christopher Satterfield, 2020
The following individuals certify that they have read, and recommend to the Faculty of Graduate and Postdoctoral Studies for acceptance, the thesis entitled:

**Automatic Identification and Description of Software Developers Tasks**

submitted by Christopher Satterfield in partial fulfillment of the requirements for the degree of Master of Science in Computer Science.

**Examination Committee:**

Gail Murphy, Computer Science  
*Supervisor*

Reid Holmes, Computer Science  
*Supervisory Committee Member*
Abstract

A software developer works on many tasks per day, frequently switching back and forth between their tasks. This constant churn of tasks makes it difficult for a developer to know the specifics of what tasks they worked on, and when they worked on them. Consequently, activities such as task resumption, planning, retrospection, and reporting become complicated. To help a developer determine which tasks they worked on and when these tasks were performed, we introduce two novel approaches. First, an approach that captures the contents of a developer’s active window at regular intervals to create vector and visual representations of the work in a particular time interval. Second, an approach that automatically detects the times at which developers switch tasks, as well as coarse grained information about the type of the task. To evaluate the first approach, we created a data set with multiple developers working on the same set of six information seeking tasks. To evaluate the second approach, we conducted two field studies, collecting data from a total of 25 professional developers. Our analyses show that our approaches enable: 1) segments of a developer’s work to be automatically associated with a task from a known set of tasks with average accuracy of 70.6%, 2) a visual representation of a segment of work performed such that a developer can recognize the task with average accuracy of 67.9%, 3) the boundaries of a developer’s task to be detected with an accuracy as high as 84%, and 4) the coarse grained type of a task that a developer works on to be detected with 61% accuracy.
Lay Summary

A software developer works on many tasks per day, frequently switching among these tasks. This constant churn of tasks makes it difficult for a developer to know the specifics of how they completed their tasks. To help a developer determine these specifics, we introduce two novel approaches. First, an approach that captures the contents of a developer’s active window at regular intervals to create representations of the work contained within a task. Second, an approach that automatically detects the times at which developers switch tasks, as well as information about the kind of task being worked on. To evaluate the first approach, we created a data set with multiple developers working on the same set of six information seeking tasks. To evaluate the second approach, we conducted two field studies, collecting data from a total of 25 professional developers.
Preface

All of the work presented in this thesis was conducted in the Software Practices Lab at the University of British Columbia and the Software Evolution & Architecture Lab at the University of Zurich. All projects and associated methods were approved by the University of British Columbia's Research Ethics Board [certificate #H12-03701].

A version of Chapter 3 has been submitted to be published: [Chris Satterfield, Thomas Fritz, Gail C. Murphy. *Identifying and Describing a Software Developer’s Tasks*]. I was the lead investigator for this work, responsible for concept formation, data collection, analysis and manuscript composition.

A version of Chapter 4 has been accepted for publication in the IEEE Transactions on Software Engineering journal: [André N. Meyer, Chris Satterfield, Manuela Züger, Katja Kevic, Gail C. Murphy, Thomas Zimmermann, Thomas Fritz. *Detecting Developer’s Task Switches and Types*]. The field studies in this chapter were designed and conducted by André N. Meyer and Manuela Züger. I assisted with the development and analysis of our machine learning approaches, as well as with manuscript composition.

Thomas Fritz and Gail C. Murphy were supervisory authors on this project and were involved throughout the project in concept formation and manuscript composition.
# Table of Contents

Abstract .......................................................... iii

Lay Summary ....................................................... iv

Preface ............................................................... v

Table of Contents .................................................. vi

List of Tables ...................................................... ix

List of Figures ...................................................... x

Acknowledgments ................................................... xii

1 Introduction ..................................................... 1

2 Related Work .................................................... 6
   2.1 Determining Intent ......................................... 6
       2.1.1 Intent from Interactions .............................. 6
       2.1.2 Intent from Documents ............................... 8
       2.1.3 Intent from a Combination of Interactions and Documents 8
       2.1.4 Finding Meaning in Artifacts ....................... 9
   2.2 Detecting Task Switches .................................... 9

3 Identifying and Describing Tasks .............................. 12
   3.1 Data Set Creation ......................................... 14
       3.1.1 Developers ........................................... 14
       3.1.2 Session ............................................... 14
   3.2 Data Collected ............................................. 15
       3.2.1 Data Annotation ...................................... 17
   3.3 Generating Task Representations ......................... 18
3.3.1 Screenshot Preprocessing ........................................ 18
3.3.2 Extracting Bags of Words .................................... 18
3.3.3 Generating Task Representations ............................ 20
3.4 RQ1: Associating Descriptions ................................. 21
  3.4.1 Gathering Task Descriptions ............................... 23
  3.4.2 Evaluation .................................................. 24
  3.4.3 Results ..................................................... 25
3.5 RQ2: Assigning Tags ............................................ 29
  3.5.1 Survey ..................................................... 29
  3.5.2 Results ..................................................... 31
3.6 Discussion ..................................................... 31
  3.6.1 Threats to Validity ........................................ 32
  3.6.2 Applying the Approach .................................... 33
  3.6.3 Artifact Access ........................................... 34
  3.6.4 Creating Vectors from Window Titles ...................... 35

4 Detecting Task Switches and Their Types ....................... 37
  4.1 Study Design ................................................ 39
    4.1.1 Study 1 – Observations ................................ 40
    4.1.2 Study 2 – Self-Reports ................................ 42
  4.2 Data and Analysis ............................................ 45
    4.2.1 Collected Data ......................................... 45
    4.2.2 Time Window Segmentation ............................. 46
    4.2.3 Task Switch Features Extracted ......................... 47
    4.2.4 Task Type Features ..................................... 50
    4.2.5 Outcome Measures ...................................... 50
    4.2.6 Machine Learning Approach ............................ 50
  4.3 Results: Detecting Task Switches ............................ 52
    4.3.1 Descriptive Statistics of the Dataset ................... 52
    4.3.2 Task Switch Detection Error Distance and Accuracy .... 52
    4.3.3 Task Switch Feature Evaluation ........................ 55
  4.4 Results: Detecting Task Types ............................... 55
    4.4.1 Identified Task Type Categories ....................... 55
    4.4.2 Descriptive Statistics of the Dataset ................... 58
    4.4.3 Task Type Detection Accuracy .......................... 58
    4.4.4 Task Type Feature Evaluation .......................... 60
  4.5 Discussion .................................................. 60
    4.5.1 Improving Task Switch Detection ....................... 60
    4.5.2 Improving Task Type Detection ......................... 61
    4.5.3 Reducing the Prediction Delay .......................... 63
    4.5.4 Applications for Automated Task Detection ............ 63
List of Tables

Table 3.1  Overview of controlled lab tasks. ........................................ 16
Table 3.2  Full task description for the App Market Research task (PrMR) as presented to developers. ...................... 17
Table 3.3  Examples of descriptions received for the Viz Library Selection (Viz) task, together with one of the expert’s ratings. .......... 24
Table 3.4  Results of mapping task representations to task descriptions written by MT workers. ................................. 27
Table 4.1  Self-reports for Study 2. ..................................................... 46
Table 4.2  Features analyzed in our study and their importance for predicting task switches and task types. ......................... 48
Table 4.3  Overview of the performance of detecting task switches, for both individual and general models. ..................... 54
Table 4.4  Overview and descriptions of the task type categories, the average time developers spent on each task type per hour of work, and the performance of our task type detection approach, for both individual and general models. .......... 57
List of Figures

| Figure 3.1 | An example of a developer working on several tasks over time, revisiting task 3 in two task segments. | 17 |
| Figure 3.2 | Process followed for the generation of task representations (the light blue boxes represent task segments). | 19 |
| Figure 3.3 | Word Clouds (WCs) generated for task segments from different tasks and developers. | 22 |
| Figure 3.4 | Accuracy comparison for the 6 different combinations of techniques used to generate task representations (TK = tokenization). | 26 |
| Figure 3.5 | Accuracy for mapping task representations to the correct task descriptions by lab developer. The dotted line represents the accuracy for a random classifier. | 29 |
| Figure 3.6 | Accuracy for mapping task representations to the correct task descriptions by MT respondent. The dotted line represents the accuracy for a random classifier. | 30 |
| Figure 3.7 | Accuracy for identifying the task based on our generated word clouds. The dotted red line indicates the accuracy for a random classifier. | 32 |
| Figure 3.8 | Comparison of performance per task with window titles as the data source vs screen content. For both approaches, results are calculated across the 12 developers with window title data available. | 36 |
| Figure 4.1 | Overview of the study design and outcomes. | 39 |
| Figure 4.2 | Screenshot of the second page of the experience sampling pop-up that asked participants to self-report their task types. | 44 |
| Figure 4.3 | Cross-validation approach for the individual models, leaving a gap of 10 samples before and after the test set to account for the dependence of samples in close temporal proximity. | 52 |
| Figure 4.4 | Confusion matrix for task type prediction. | 59 |
Figure A.1  An example screenshot captured with our tool of a developers inbox at the start of a data collection session. . . . . . . . . . 84
I would like to thank Gail Murphy and Thomas Fritz for fuelling and at times reining in my intellectual curiosity. This work would not have been possible without their guidance and support.

I would also like to thank my co-authors on the TSE paper which makes up the content of Chapter 4, and in particular André N. Meyer for their invaluable contributions.

Last but not least, I would like to thank my parents and my partner Ashley Wong for encouraging and supporting me throughout this project.
Chapter 1

Introduction

Software developers work on many tasks in a day, switching between them constantly [22, 44]. This constant switching, and the variety and high number of tasks, make it difficult for developers to keep track of which task they worked on when. Yet, developers can benefit from this information in several ways. For instance, knowing when and how long a task was worked on can help in the tracking of time spent on tasks, aiding planning, retrospection and reporting activities (e.g., [42]). As another example, knowing what information is accessed as part of the task can help to recall what information is needed when a task is resumed [29, 56, 58].

Some developers, as they work, manually track and note which information they access while performing a task as a form of externalization of the working state of a task [58]. This saved information can help a developer resume the same task later or can serve as a means of knowing which task was worked on when. This manual approach is time consuming and requires substantial effort from the developer. The Mylyn tool seeks to reduce this burden by enabling task descriptions in a development environment to be activated with a click of a button, after which the tool can automatically track the information a developer accesses and can associate it with the task [29]. When a developer returns to a task and reactivates the description, Mylyn can re-populate the environment with the information previously accessed as part of the task. However, Mylyn still requires the developer to manually write task descriptions, and developers must remember to activate and deactivate the task when work begins or stops on the task.
In this thesis, we explore how to move towards a fully automatic solution to supporting developers in managing the contexts of their tasks. To this end, we consider four research questions:

**RQ1:** Can we automatically associate descriptions of developer’s tasks with information they access as they work?

**RQ2:** Can we automatically assign tags to information accessed by a developer to describe the task being performed?

**RQ3:** Can we automatically detect the times at which a developer switches between tasks?

**RQ4:** Can we automatically determine the types of the tasks that a developer works on?

Approaches which can help to address these research questions would make it easier for developers to find and resume previous work (RQ1, RQ2, RQ4), enable us to tailor support systems depending on the type of task being performed (RQ4), and contribute to the improvement of existing task support systems by the relaxing of constraints (RQ1, RQ2, RQ3).

To explore these research questions, we developed and evaluated two approaches in separate studies. We first consider RQ1 and RQ2, developing an approach to generate vector and visual representations of a task using information extracted from a developer’s screen content. In developing this approach, we made the assumption that the times at which a developer switches tasks were known, in order to simplify an otherwise intractable problem. We next considered RQ3 and RQ4, developing an approach to automatically detect the task switches of a developer, and to detect the type of task a developer works on. In comparison to the first approach we developed, this approach uses less data and a less invasive approach to data collection, but produces a more coarse grained description of the content of a task.

We begin by exploring how to support developers by automatically identifying the tasks they work on. To this end, we developed an approach that continuously captures the screen of a developer and retrospectively generates, based on given task switch information, representations of the tasks on which a developer worked.
These representations can be used to help determine and describe the tasks that a developer performed during particular time periods. Specifically, our approach utilizes optical character recognition (OCR) to extract information directly from the developer’s screen content. Using OCR allows us to capture only the relevant sections of the resources a user is viewing, namely the parts that the developer can see and interact with. From the OCR output, we apply natural language processing and information retrieval techniques to generate a vector representation of the task on which a developer worked. We experiment with a variety of techniques for generating vector representations of tasks as part of our investigations. Using the same techniques, we also generate a visual representation in the form of a word cloud that highlights the most relevant words that describe the task. Word clouds have been shown in previous works to be useful for aiding users in determining the topic of a document [23]. An advantage of our approach is that it is agnostic to the applications a developer is using to perform the work.

In order to evaluate the efficacy of our approach to generate vector and visual representation generation, we applied these techniques to a data set we generated in a controlled lab setting with 17 developers. We then conducted two separate surveys, one in which we asked respondents to summarize the work which the developers performed in the lab, and one in which we asked respondents to match our word cloud representations to the task from the controlled lab setting which generated them. Across all task segments in our data set, we found we were able to associate task descriptions provided by our respondents correctly in 70.6% of cases, and that respondents were able to match a generated word cloud to the task segment correctly in 67.9% of cases.

A key assumption we made in the development of this approach is that the times at which developers switch tasks are known. We made this decision in order to give our approach the best possible scenario for successful performance, ensuring each task segment contains only information relevant to that task. But to move towards a fully automatic solution for supporting developers tasks, it is imperative that we are able to detect task switches automatically. Previous works have attempted to automatically detect task switches (e.g. [30 48, 55, 74, 75]). However, the evaluations performed of these techniques have been limited and the results have been poor in terms of prediction accuracy. In addition, many of these approaches
that are specific to the software engineering domain focus on detecting switches within the IDE, meaning only development related work can be captured. Such work makes up a small portion of the actual work developers perform in a day [43]. To improve the practical applicability of our approach to addressing RQ1 and RQ2, we sought to improve upon the results of these previous approaches to task switch detection by developing our own machine learning approach. We also considered the problem of classifying the type of task that a developer is working on (e.g., development, awareness, planning, etc.). Such an approach to understanding the context of a task could help us to better support developers as they work, as well as enabling detailed retrospective time tracking. Using features extracted from developers’ computer interaction information, we explored various machine learning methods to develop an approach for automatic task switch and type detection. We evaluated this approach in two separate field studies, achieving an overall accuracy of task switch prediction of 84% and 61% accuracy for type detection.

Overall, our results show promise for being able to further automate task support for software developers performing information seeking tasks. This thesis makes five contributions:

- An application-agnostic approach to associate descriptions of information seeking software development tasks with periods of time on which a developer worked on the task, including an evaluation of the efficacy of the approach on a data set from a controlled lab setting.
- The evaluation of a variety of techniques for generating vectors describing information accessed as a developer works for the purpose of associating the information with tasks.
- An evaluation of the use of word clouds to describe work performed in segments of time that enables developers to recognize work belonging to a particular task.
- An approach to automatically detect task switches and types based on developers’ computer interaction that is not limited to the IDE.
- An evaluation of our approach to task switch and type detection on data gathered from a field study with 25 professional developers.
We begin by comparing the problems we tackle and the approaches we investigate to previous work in this area for software development in Chapter 2. We then describe the data collection, representation generation, and evaluation of our approach for generating task representations in Chapter 3. Next we present our approach to task switch and type detection and their evaluation in Chapter 4. We discuss how these approaches complement each other and directions for future research in Chapter 5. Finally, Chapter 6 concludes the thesis.
Chapter 2

Related Work

Work related to our research can roughly be categorized into work that focuses (a) on determining the intent of developers’ actions based on their computer interactions and the documents and artifacts they access or produce, and (b) on detecting developers’ task switches.

2.1 Determining Intent

The research questions we pose require some level of understanding of the intent of a developer in undertaking particular work. Determining the intent of a developer is a growing area of research. The more we know about a developer’s intention, such as the task she is working on, the better we can support the developer, for example by providing better code recommendations (e.g., [16, 30]). Approaches have been developed to determine intent from how a developer interacts with the computer, from the documents produced by a developer and from a mix of both. We describe approaches in each of these categories and also describe related work in finding meaning in artifacts.

2.1.1 Intent from Interactions

For some research systems, intent is specified through specific interactions a developer takes within the environment in which they work. In the Mylyn system, a developer can indicate through an explicit click of the button on which issue they are
currently working: the text in an issue provides information about the developer’s intent [29]. In the Jasper system, a developer can create special working areas of their development environment into which fragments of work can be placed for later recall [11]. The approach we consider in this thesis relieves the developer from a priori indicating work on a specific task.

Other researchers have attempted to determine automatically the higher-level activities developers perform based on their interaction with the computer. For example, Mirza et al. used temporal and semantic features based on window interactions and the window titles over 5 minute time windows to predict one of six work activity categories: writing, reading, communicating, system browsing, web browsing, and miscellaneous [50]. In a controlled lab study and field study with 5 participants, they achieved an accuracy of 81%. Koldijk et al. investigated the predictive power of keyboard and mouse input, as well as application switches and the time of day, for predicting a larger set of 12 high-level task types—such as reading email, programming, creating a visualization—for a given 5 minute period of time [33]. Using classifiers trained on an individual basis, they were able to achieve up to 80% accuracy. However, they found that a classifier trained on one person is highly individual to that person and does not generalize well to other people. In an approach more specific to software developers, Bao et al. explore the use of conditional random fields (CRFs) to predict one of six development activities: coding, debugging, testing, navigation, search, or documentation [5]. Applying their approach to data collected from 10 software developers over a week, the authors found they were able to classify an activity with an accuracy of 73%. The results of Bao et al. point to the difficulty of determining at a fine granularity what a developer is working on at a specific moment. In our work, with our first approach we aim to determine the content of a developer’s task rather than the kind of activity being undertaken. We also attempt in our second approach to improve on the results of the previous authors in determining the coarse grained type of the task that a developer is working on, using less data and a less invasive approach to data collection than the one used in our first approach.
2.1.2 Intent from Documents

Researchers have also looked into the extraction of intent from natural language documents associated with a software development. Early on, researchers have tried to detect the coarse intent of sentences in emails and tried to summarize them, for example to add them to a to do list (e.g., [14]). Di Sorbo et al. introduced the concept of intention mining in the context of emails in software development [79]. They used a Natural Language Processing (NLP) approach to classify the content of development emails according to the purpose of the emails, such as feature request or information seeking. The researchers defined six categories that describe the intent of a developer’s sentence and reported a 90% precision and 70% recall for their approach in the context of email intent classification. Huang et al. attempted to generalize the approach of Di Sorbo et al. to developer discussions in other mediums, for example those contained in issue reports [25]. They found that the NLP patterns used did not adapt well to other mediums, achieving an accuracy of only 0.31. By refining the taxonomy of intentions defined by Di Sorbo et al., and applying a convolutional neural network (CNN) based approach, the authors were able to improve on the results of the original paper by 171%. These approaches aim to classify what the content of a document is attempting to state as compared to our approach in this thesis which aims to determine what the developer is attempting to do.

2.1.3 Intent from a Combination of Interactions and Documents

Shen et al. [75] use a combination of information about how a user interacts with windows on their screen and email messages the user handles in their TaskPredictor system. Using supervised machine learning, they predict on which task a user is working. However, this techniques requires the user to pre-define the tasks on which they work so that they can be predicted and the classifier needs to be trained on some of the user’s data beforehand. Our approach differs in assessing methods for representing the work being performed based on information that a developer works on through screen scraping; these representations can be used for predicting which of a known set of tasks the work represents and for generating visual representations of the work that a developer can recognize irrespective of having a set of known
tasks.

2.1.4 Finding Meaning in Artifacts

The content of artifacts created as part of, or about, software development contain significant meaning. Software engineering researchers have developed techniques to find particular meaning in artifacts that have similar characteristics to the approach we develop in this thesis. For example, Ponzanelli et al. present CodeTube, an approach that mines video tutorials from the web to enable developers to query the contents of the tutorial to retrieve relevant fragments [62]. The authors used OCR and speech recognition in order to extract text from the videos and evaluate the relevancy of fragments to the user’s query. The determination of what a segment of video is about is similar to the problem we tackle of what a segment of a developer’s work is about.

2.2 Detecting Task Switches

Several researchers have explored the detection of task switches mostly for general knowledge workers. These approaches mainly differ in the features they used to identify the task boundaries or switches, ranging from semantic features to temporal features, the method they use, unsupervised versus supervised, and the way they evaluated their approach. One of the most prominent approaches is by Shen et al. [73, 74, 76, 81] that is mainly based on semantic features and supervised learning. They reused an approach, TaskTracer [18], that allows users to manually indicate the tasks they are working on, and additionally tracks their application interactions in the background, including window titles. Based on the assumption that windows of the same task share common words in their titles, they create vectors from window titles and identify task switches based on a textual similarity measure using the users’ previously declared tasks and supervised learning. After the first version [73], they further improved their approach to reduce the number of false positives and to be able to predict task switches online [74, 76]. Their evaluation is based on a small set of two users and counts a task switch as accurate if it falls within a 4 to 5 minute time window of a real switch, which is a very coarse measure, given the frequent task switching in today’s environment that happen every few minutes [22 43].
Based on the assumption that switches between windows of the same task occur more frequently in temporal proximity than to windows of a different task, Oliver et al. [55] examined a temporal feature of window switches within a 5 minute time window in addition to semantic features and using an unsupervised approach. An evaluation based on 4 hours of a single participant, showed a precision of 0.49 and recall of 0.72. Researchers have also used other temporal features, in particular, the frequency of window events, to determine task switches. Under the assumption that users navigate between windows more frequently when they switch tasks, as opposed to during a task, Nair et al. [51] developed a system that calculates window event frequency based on fixed 5 minute time windows. An evaluation with 6 participants resulted in an accuracy of 50%. Mirza et al. [48] relaxed the constraint of a fixed time window, used adjusted frequency averages and studied the various approaches with 10 graduate students. They found that their approach improved the accuracy and achieved an overall accuracy of 58%. Overall, previous research has shown that detecting task switches is difficult, even for short periods of time and in controlled environments. In our work, we focus on software development work and extend these approaches by including and examining both, semantic and temporal features of window events, as well as user input features, and by conducting two studies with professional software developers.

Little research has been performed on task switch detection in the software development domain and all of this research has focused solely on software development tasks within the IDE. One of the first, Robillard and Murphy [69] proposed to use program navigation logs to infer development tasks and they built a prototype, without evaluating it. In 2008, Coman and Sillitti [13] focused on splitting development sessions into task-related subsections based on temporal features of developers’ access to source code methods and evaluated their approach with 3 participants over 70 minutes each, finding that they can get close to detecting the number of task switches, yet the point in time when the task happens is a lot more difficult. Zou and Godfrey [89] replicated Coman and Sillitti’s study in an industrial setting with six professional developers and found that the algorithm detects many more task switches than the ones self-reported by the participants with an error of more than 70%. Finally, on a more fine-grained level, Kevic and Fritz [30] examined the detection of activity switches and types within a change task using semantic,
temporal and structural features. In two studies with 21 participants, they found that activity switches as well as the six self-identified activity types can be predicted with more than 75% accuracy. In contrast to these approaches, we focus on all tasks a developer works on during a day, not just the change tasks within the IDE.
Chapter 3

Identifying and Describing Tasks

In this chapter we focus on the problem of whether the topic of work—a task—can be automatically identified based solely on the information that a software developer is accessing as part of a task. To simplify the investigation of this problem, we assume for the purposes of the research in this chapter that the times at which developers task switches are known, simulating a developer manually marking task switches as they work. We consider two research questions:

RQ1: Can we automatically associate descriptions of developer’s tasks with information they access as they work?

RQ2: Can we automatically assign tags to information accessed by a developer to describe the task being performed?

Approaches that can help address the first question would enable a developer to locate when work was being performed on a particular known task. These approaches could help a developer look back in the history of their work to identify information accessed as part of a known task or could be used to help complete reports on time spent on particular known tasks. These approaches could also help identify which task a developer worked on when, relaxing constraints in tools like Mylyn [29] or Jasper [11] for a developer to indicate when work begins or ends on a task. Approaches that can help address the second question would further relieve constraints associated with having to know the tasks being performed. Instead, a
developer could access tags that describe the work from which a task description could be written or associated.

To evaluate the ability of our approach to address the two research questions, we required a data set. To generate this data set, we had 17 experienced software developers work on six tasks in a controlled lab setting. We designed the six tasks to be representative of information seeking tasks often performed by software developers. Previous studies have shown that developers spend 31.9% of their day on such tasks [21]. Each task description consisted of a paragraph or more of text indicating what information was needed on a given topic. More information about the tasks developers worked on can be found in Chapter 3.1. As a developer worked, we recorded their screen and took notes about when they worked on each task. We interrupted developers and prompted changes in the order in which tasks were performed to gather more realistic data that involves task switching. From this data, we generated representations of the tasks using our approach. We make this data set available to other researchers so they may build on it to investigate other approaches.

To investigate the first research question, we evaluated a variety of techniques for generating representations of the tasks from the data to compare to descriptions of the tasks. To reduce bias in the descriptions of tasks, we gathered multiple short summaries for the tasks using professional IT personnel sourced through Mechanical Turk. Our evaluations revealed that a simple approach using TF-IDF yielded the best results (Accuracy: 70.6% for task segments and 75.5% for all segments of a task).

To investigate the second research question, we used the best technique identified from our evaluation of the first research question (TF-IDF with word tokenization) to generate word clouds for task segments in our dataset. We then conducted a survey in which 28 software developers had to match a randomized subset of the generated word clouds to the original tasks. Overall, the respondents in this survey were able to identify the correct tasks for the word clouds in 67.9% of the cases.
3.1 Data Set Creation

To support the investigation of the two research questions, we created a data set from 17 developers working in a controlled laboratory setting on a set of six tasks over a 2 hour time period. We chose a laboratory setting to be able to gather data from multiple developers working on the same tasks. The full data set is available online [72].

3.1.1 Developers

We recruited 17 participants—that we will call developers in the following—through advertising at our university and personal contacts. Of these developers, 10 were graduate students, 4 were upper year undergraduates, and 3 were interns at a mid-sized software company. All developers had several years of experience in software development, with an average of 6.4 (±2.4) years per developer. 10 of the developers were female, and 7 were male. All developers were residents of Canada.

3.1.2 Session

At the start of a session, a developer was presented with a list of 6 tasks to perform within a 2 hour time period. The tasks were presented in the form of unread emails sitting in an inbox accessed by a webmail client. Figure A.1 shows a screenshot of the inbox at the start of a session. The order in which the tasks appeared in the inbox for a developer was randomized. The tasks represent a variety of non-coding tasks associated with software development that commonly form part of software development. We chose to focus on non-coding tasks because the developers could attempt more tasks in a two hour block compared to coding tasks that would require gaining familiarity with a codebase and because it broadened the approaches that could be used to extract meaning from the information accessed as part of a task by a developer. We discuss the implications of this choice further in Chapter 4.6.

Table 3.1 provides a short description of each of the six tasks, including a short name that we use in this chapter to refer to a specific task; the short task name and description was not presented to developers. An example of one of the actual task descriptions used in this study is presented in Table 3.2. The full task descriptions for each task can be found in Appendix A. We intentionally designed
the App Market Research Task and Recommend Tool Task as tasks which were likely to have very similar information accessed as part of working on the task to allow us to assess the discriminative power of our approach.

We asked a developer to work on the tasks on a laptop with a 13.3 inch, 1440x900 sized screen running macOS which was instrumented with our own recording tool, PersonalAnalytics

1\[42\]. As a developer worked on the tasks, the tool recorded screenshots of the developer’s active window at 1 second intervals. Application names and window titles were also recorded whenever they changed. To simulate interruptions, the tool produced a popup in random intervals lasting from 6.5 to 16.5 minutes which asked the developer to switch to a new task. The average time between popups was selected as 11.5 minutes, in accordance with Gonzlez and Mark’s findings on the average amount of time knowledge workers spend in a working sphere segment before switching \[22\]. To simulate the disruptive effects of a real external interruption, we also asked developers to solve an arithmetic question before switching to a new task. These popups were excluded from our tools recordings to avoid biasing our results.

As a developer worked on the tasks, a researcher manually annotated the times at which the developer switched tasks, also keeping track of the task the developer was working on. After the session was complete, the times at which switches happened were verified and adjusted by reviewing a screen capture that ran in the background of the provided laptop, to ensure highly accurate task switches were recorded.

### 3.2 Data Collected

In total, we were able to collect screenshot data for all 17 developers and on all six tasks for each developer. Also, all but one developer completed the 6 tasks within the allowed time period. On average, developers took 91.2 ±17.5 minutes to complete the six tasks and we collected an average of 5131 screenshots per developer. Due to a technical issue, we were able to gather window titles for only 12 of the 17 developers.

1\[https://github.com/sealuzh/PersonalAnalytics/tree/mac\]
Table 3.1: Overview of controlled lab tasks.

<table>
<thead>
<tr>
<th>Abbrev.</th>
<th>Short Task Name</th>
<th>Short Task Description (by us)</th>
</tr>
</thead>
<tbody>
<tr>
<td>BugD</td>
<td>Duplicate Bug Task</td>
<td>Examine a collection of bug reports from a Bugzilla repository to determine if any were duplicates.</td>
</tr>
<tr>
<td>Viz</td>
<td>Viz Library Selection Task</td>
<td>Research visualization libraries and identify one which is suitable for outlining the benefits of your companies tool, for creating a presentation to clients.</td>
</tr>
<tr>
<td>PrMR</td>
<td>App Market Research Task</td>
<td>Perform market research on three productivity apps. Identify common functionalities, similarities and differences, and report on your findings.</td>
</tr>
<tr>
<td>PrRec</td>
<td>Recommend Tool Task</td>
<td>Examine app store reviews for three productivity apps (the same ones as above) in order to recommend one to your coworker.</td>
</tr>
<tr>
<td>DeepL</td>
<td>Deep Learning Presentation Task</td>
<td>Prepare in advance answers to likely questions for a hypothetical presentation you are giving about potential deep learning applications.</td>
</tr>
<tr>
<td>BIC</td>
<td>Blockchain Expert Task</td>
<td>Answer your coworkers follow-up questions about a hypothetical presentation you gave about the different ways your company could make use of blockchain.</td>
</tr>
</tbody>
</table>
Table 3.2: Full task description for the App Market Research task (PrMR) as presented to developers.

The software company you work for is considering expanding into the productivity tool sphere. Your manager has asked you to do some market research on 3 of the most popular already existing apps in this domain: Microsoft To-do, Wunderlist, and Todoist. Provide a short written summary of the similarities and differences between these 3 apps.

![Diagram](image)

Figure 3.1: An example of a developer working on several tasks over time, revisiting task 3 in two task segments.

3.2.1 Data Annotation

Using the task annotations collected by the researcher during each session, we annotated the collected data with the task switches and the task the developer was working on. Each session with a developer resulted in a developer working in an interleaved fashion on the six tasks. Figure [3.1](image) depicts a portion of a developer’s work, showing an example of the interleaving. We define a task segment as the period of time between two task switches, during which a developer was working on a specific task. We define a task segment grouping as the collection of all task segments that collectively represent work on a specific task. We use task segment groupings as a baseline for evaluating our approaches, as it represents the simplest possible scenario in which we have the entirety of the information accessed during the work on a task available to us for our analysis.
3.3 Generating Task Representations

The notion of a task is an abstract concept. In order to interact with tasks in a meaningful way, we must be able to represent tasks in a concrete manner. We generate two types of task representations: vector space representations (vectors) and visual representations (word clouds). These representations are created from the screenshots of the active windows that we gathered in the data collection phase using a series of extraction and processing steps. An overview of the steps involved is depicted in Figure 3.2.

3.3.1 Screenshot Preprocessing

To prepare the collected screenshots of the active windows for the optical character recognition (OCR) with Tesseract [82], we preprocess the screenshots in accordance with suggested best practices for the Tesseract tool. Specifically, we convert the colored screenshots to grayscale and scale the resolution down to 300 DPI. These steps are recommended as Tesseract was originally intended for reading paper documents (i.e., black on white). In addition, since most application windows have bars, such as a menu or a bookmark bar, at the top of the window, and these bars generally do not contain information specific to the task at hand, we consider it as noise and crop a percentage of the top of the screenshot to remove this noise. Through experimentation, we found that removing the top 15% of the screen across all application window screenshots provides the best balance between removing noise without much loss of meaningful content for the data that we collected. Note that this percentage might have to be adjusted when working on a screen with a different resolution and size. All screenshot preprocessing steps were automated with the ImageMagick tool [26].

3.3.2 Extracting Bags of Words

After preprocessing, we used the Tesseract OCR engine to extract the textual content of each screenshot. Tesseract tries to also preserve the format of the text, and produces a structured string for each screenshot that we store in a document, one for each screenshot. Note that the structured strings produced by Tesseract still contain substantial noise even after the preprocessing. For instance, an ‘I’ could
be misinterpreted as the number ‘1’ or the letter ‘l’. As well, many nonsensical artifacts were produced due to noise from items like images and menu bars on the screenshot.

To break up these documents into usable pieces of information (words) and reduce some of the noise, we applied one of two techniques, either (a) tokenization, or (b) keyword extraction. We chose tokenization since it is a common practice when processing natural language text, and we chose keyword extraction as an alternative,
since it might help us to reduce some of the noise in the extracted text. For the
tokenization, we used the Natural Language Toolkit (NLTK) [54] Version 3.2.5 and
applied standard word tokenization techniques based on whitespace and punctuation
to generate lists of tokens containing all of the words in a screenshot. To further
reduce the noise, we then performed a dictionary check and discarded any tokens
that were not correct English words. Further, we removed all stop-words from the
tokens. For the keyword extraction, we used an open source implementation [64]
(version 1.0.4) of the RAKE algorithm[70]. Based on an input string, RAKE
produces a set of keywords that is equal to 1/3 of the number of original words (not
counting duplicates).

After breaking up each document into a set of words, we stemmed all the words
using the Porter stemmer implementation from NLTK. Finally, we created a bag of
words—that is, a record of the frequency of each word—for each task segment by
aggregating all words extracted from all screenshots of a task segment. In order to
produce baseline task representations from all information on a task (task segment
grouping), we also created bags of words by aggregating all words from all task
segments belonging to the same task.

### 3.3.3 Generating Task Representations

A bag of words is itself a primitive representation of a task where the words can
be seen as words that describe the task and the frequency of each word in the bag
as the importance of it. However, this representation only reflects importance of a
word with respect to the current document (task segment). To also take into account
the relevance of the word in context of the overall work of the developer and further
help filtering the noise from the screenshots and the OCR, we experimented with
several natural language processing (NLP) and information retrieval techniques.
Since TF-IDF performed best in our comparison (see Chapter 3.4), we present the
generation of task representations using TF-IDF in the following.

The formula for TF-IDF is defined as $f_i \times idf_i$ where $f_i$ represents the frequency
with which a term $t$ occurs within a single document, and $idf_i$ measures the inverse
document frequency of $t$. The value of $idf_i$ is calculated as $\log \frac{N}{df_i}$, where $N$ is
the total number of documents and $df_i$ is the document frequency of $t$, or more
specifically the number of documents that the term $t$ appears in.

For our purposes, we consider a document in this scenario as the bag of words generated from each task segment $ts$, and $idf_i$ is calculated based on the collection of all documents from the same developer. Note that for our baseline in which we do not distinguish between task segments but group all segments of a task together, we consider these groupings as the documents and the six documents (one per task) as the whole corpus for TF-IDF.

Using the TF-IDF scores for each word in a bag of words, we ranked the words found within each task segment from most to least relevant. We then used this list to produce (a) a visual representation in the form of a word cloud WC, and (b) a vector space representation $V$.

In generating the word cloud, we found that a task contained on average 1148 unique words after the removal of stop-words. Since the majority of the words are not strictly relevant to the task itself, we limited the selected words to the 100 most relevant words. We found a 100 word limit to achieve a balance between not showing relevant words and cluttering the visualization with irrelevant information. We further used the TF-IDF relevancy score of the words to determine the size of the word. Figure 3.3 shows two examples of such word clouds. For the vector space representation, we formed multi-dimensional vectors with one dimension per unique word in the set of all words from all documents for a developer. Each task segment can then be represented as a vector that has a non-zero entry for each word in the bag of words that represent the task segment, and the non-zero entry being the TF-IDF score of the word.

3.4 RQ1: Associating Descriptions

Our first research question asks whether we can automatically associate descriptions of a developer’s tasks with the information the developer accesses as she works. Performing this association automatically is challenging because there are many ways in which a developer can complete a task and there are many ways in which a developer can describe the task on which they are working.

The data we collected in the lab setting (Chapter 3.1) includes a number of ways in which the tasks assigned could be completed. While participants in the lab
setting had some overlap in the resources they accessed as part of a task, no two participants completed a task in exactly the same manner.

Similarly, developers are likely to tailor their task descriptions towards the ways they might approach a task. To study the first research question, we therefore also needed a range of descriptions of the tasks on which the developers had worked. To gather these descriptions, we employed Amazon’s Mechanical Turk. Given a range of descriptions collected in this way, we are able to assess how the range of techniques we developed for generating task representations (Chapter 3.3) can address the first research question.
3.4.1 Gathering Task Descriptions

To capture a range of task descriptions, we distributed a survey via Amazon’s Mechanical Turk. As a requirement for responding to our survey, we asked that respondents be currently or previously employed in the software industry. In total we received responses from 29 respondents. These respondents represented a range of fluency with English and a range of experience in software development. On average, respondents had 6.2 ($\pm 5.4$) years of software development experience, and 3.8 ($\pm 3.5$) years of professional development experience. Of these respondents, 24 reported that they were native English speakers, while 3 reported being fully fluent and 2 reported being proficient.

Respondents of this survey were presented with the same set of six full task descriptions that we also used for the data set creation in the controlled lab setting. An example can be seen in Table 3.2. We asked respondents to “Please summarize the task described below in your own words, as you might write it for your own reference in a to-do list or similar. Please limit your response to at most 15 words.”. Thereby, we randomized the order in which the full task descriptions were presented.

To filter out irrelevant or low quality responses, we asked two external experts to rate the quality of every task description generated by each respondent. Each rater used a scale from 1-3 to indicate the relevancy and quality of the responses, with a score of 1 indicating an irrelevant response, 2 indicating relevant but low quality responses, and 3 representing relevant and high quality responses. We found that the distinction between responses rated 2 or 3 varied greatly between our two experts, but that there was a strong consensus with regard to the responses which were rated 1 / irrelevant (Cohen’s Kappa: 0.74, indicating strong agreement [41]). These irrelevant responses tended to come in multiples from the same participants. We removed all participants with irrelevant responses and considered only those responses which both authors rated with a score of 2 or higher, leaving us with 20 participants and a total of 120 task descriptions. A sample of 3 responses for one task with the ratings by one expert rater is depicted in Table 3.3.
Table 3.3: Examples of descriptions received for the Viz Library Selection (Viz) task, together with one of the expert’s ratings.

<table>
<thead>
<tr>
<th>Rating</th>
<th>Task Description (Survey Response)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Irrelevant (1)</td>
<td>I would suggest SIMILE Exhibit or InfoVis Toolkit for Javascript libraries to create a visualization.</td>
</tr>
<tr>
<td>Low Quality (2)</td>
<td>Visualize workers work pattern.</td>
</tr>
<tr>
<td>High Quality (3)</td>
<td>Create visualizations for product benefits. Select libraries and give existing work examples.</td>
</tr>
</tbody>
</table>

3.4.2 Evaluation

From the controlled lab setting, we have 189 task segments and 102 task segment groupings. From the Amazon Mechanical Turk survey, we have 20 descriptions for each task, resulting in a total of 120 task descriptions. We wish to determine if the approach we developed for generating task representations can be used to determine which task segment (or task segment group) maps to which task description with sufficient precision and recall, even when these task descriptions might vary. We also wish to determine which choice of techniques within the approach gives optimal results. As our ground truth, we use the annotations made by a researcher during the data set collection phase which tell us what task description a specific task segment or task segment group correctly maps to.

Our evaluation consists of considering each task segment from a lab developer’s work and mapping it to one of the six task descriptions produced by a Mechanical Turk respondent. We use this evaluation method that assumes a complete set of descriptions as we wish to assess how well our approach might work in a situation where a developer may be trying to determine, from a given set of tasks, when they performed work on each task. For the mapping, we generate a vector space representation of the task segment $V_{TS}$ as well as one for each of the six task descriptions $V_1$ to $V_6$ produced by a respondent and then calculate the cosine similarities between $V_{TS}$ and each of $V_1$ to $V_6$. We choose the task description most similar to our generated task representation and evaluate it by comparing it to the ground truth to determine if it is correct.

For generating task representations from task segments in vector space format,
we experimented with and compared six (3 x 2) different combinations of techniques: 3 different techniques for vectorization (term frequency, TF-IDF, and word2vec word embedding), and 2 different techniques for extracting bags of words (tokenization using NLTK, and keyword extraction using RAKE as described in Chapter 3.3.2). TF-IDF vectors were generated as described in Chapter 3.3.3, while TF vectors were generated directly from the bag of words calculated from an individual task segment without taking into account inverse document frequency, and only scaling by TF. For the word embedding vectors, we used the Gensim\textsuperscript{2} implementation of Word2Vec \cite{46}, training on a Wikipedia corpus collected in 2018. Specifically, we created a word2vec vector for each word in the bag of words representing a task segment, and then averaged all these vectors to generate a single word2vec vector for the task segment.

To generate vectors from the task descriptions of Mechanical Turk (MT) workers, we tokenized the task descriptions using NLTK (keyword extraction is not useful in this case given the brevity of the descriptions), and then applied the exact same vectorization technique as used for the task segments, i.e. either TF, TF-IDF, or word2vec word embedding. TF-IDF vectors were generated on the fly during our evaluation, as the IDF calculation varies depending on the document set (i.e., the task segments) of the developer dataset being evaluated.

3.4.3 Results

Figure 3.4 illustrates the results of the comparison between the six different combinations of vectorization and word extraction techniques. Overall, the combination of TF-IDF with simple word tokenization performed the best, however the differences are small compared to the combination with RAKE or using just TF. Ultimately, word2vec performed the worst for the generation of task representations and mapping to the task descriptions. Since word2vec is also the most computationally intensive, it was the least appropriate for this scenario. Based on these results, we selected the combination of TF-IDF with word tokenization (NLTK) as the approach that we use for the remainder of the chapter.

Table 3.4 presents the results of the evaluation when mapping task representa-
Figure 3.4: Accuracy comparison for the 6 different combinations of techniques used to generate task representations (TK = tokenization).

We compare the precision and recall for each of the 6 tasks, calculated on a per task segment basis (or with the baseline of the per task segment grouping).
Table 3.4: Results of mapping task representations to task descriptions written by MT workers.

<table>
<thead>
<tr>
<th>Task Segments</th>
<th>Task Segment Groupings</th>
</tr>
</thead>
<tbody>
<tr>
<td>BugD</td>
<td>Viz</td>
</tr>
<tr>
<td>Precision</td>
<td>92.6%</td>
</tr>
<tr>
<td>Recall</td>
<td>82.3%</td>
</tr>
</tbody>
</table>
Overall, using only task segments, our approach achieved high accuracy across all tasks (70.6%) in comparison to a random classifier (16.7%). Accuracy for task segment groupings was moderately better (75.5%). This is a promising result as it indicates that there is often already sufficient information in an individual task segment to predict the task that is being performed; adding more information helps some but does not make a dramatic difference.

Our approach performed well at predicting tasks with a distinct focus, such as DeepL and BIC, with precision values over 80%. This result is unsurprising, as in order to perform these tasks, the developers in the lab setting tended to turn to resources that contained a dense amount of highly specific information related to these topics, such as the Wikipedia pages for blockchain and deep learning. The presence of dense, consistent information eases the production of accurate representations for the tasks. Our approach also performed well at recognizing the BugD task with precision over 92%. We found this result surprising as we expected this task to be one of the more difficult tasks to predict, especially since our summary authors were given no information about the content of this task, beyond that it involved finding duplicate bugs in a Bugzilla repository. As expected, the most difficult to predict tasks were the PrMR and PrRec tasks. These tasks are very similar and as such, resulted in very similar task representations as well as very similar task descriptions and in turn, in a high confusion between the two tasks.

Figure 3.5 and Figure 3.6 illustrate the results broken down on a per developer and a per MT respondent level. Despite differences in the way each developer performed each task, the results are fairly consistent across developers, ranging from a minimum accuracy of 62.9% to a maximum of 77.3%. Across the MT respondents that authored the task descriptions, the results are mostly consistent, however, there is a significant variation for a few respondents. The respondents for which the accuracy of mapping task representations to their task descriptions were rather low tended to be ones who authored multiple descriptions that were also rated lower by the experts (e.g., item 2 in table 3.3 was written by author S6). These result demonstrate that the task representations are relatively robust across developers and different ways of performing the tasks, and that writing precise and somewhat detailed descriptions of the tasks being performed clearly impacts the results of our approach.
3.5 RQ2: Assigning Tags

To address the question of whether we can automatically generate tags to information accessed by a developer which would help the developer to identify what task she worked on during a specific period of time, we evaluated the word clouds we generated as described in Chapter 3.3.3. Specifically, we asked 28 participants experienced in software development to match our generated word clouds to the original full task descriptions of the tasks that were performed during the data set creation.

3.5.1 Survey

To evaluate the quality of our automatically generated word clouds as a visual representation of a task, we conducted a survey with experienced software developers. Participant were recruited through personal and professional contacts, and as an incentive for responding were entered into a draw for one of two $25 gift cards if they desired. In total, we received survey responses from 28 individuals, with an
average of 8.0 (±3.9) years of software development experience. 20 participants were male, while 8 were female. 9 participants reported they were native English speakers, 12 reported that they were fully fluent in English, and the remainder (7) reported that they were proficient in their understanding of English.

We asked our participants to match word clouds to corresponding tasks by presenting them with the list of the six full task descriptions that we also used for the data set creation. An example of one of the descriptions can be seen in Table 3.2. The word clouds used in the survey were generated following the procedure described in Chapter 3.3.3 Using the data that we collected across all 17 developers in the data set creation (Chapter 3.1), we randomly selected 4 task segments and 4 task segment groupings for each of the six tasks and generated word clouds for these, resulting in a total of 48 word clouds. Since asking survey participants to examine a total of 48 word clouds would be too much and impractical, we randomly selected and asked each participant about 12 of the 48, ending up with 2 word clouds (1 for a task segment, 1 for a task segment grouping) for each of the six tasks. Examples of
two of these word clouds can be found in Figure 3.3. We asked participants to read the six full task descriptions and to then identify which task the presented word clouds describe best. Participants also had the option to indicate that the word cloud does not match any task.

3.5.2 Results

We aggregated the results of the survey responses to obtain accuracy ratings for the word clouds we generated. Overall, the average accuracy of mapping word clouds to the corresponding tasks was 67.9% for the word clouds generated from task segments, and 69.6% for the word clouds generated from groupings. Figure 3.7 shows the breakdown of the accuracy on a per task level. The success rates of our participants varied widely between tasks. For example, for the blockchain expert task BlC, our participants were able to correctly identify the task for the generated word cloud 100% of the time. Conversely, participants struggled to properly identify the task for the word clouds generated for the duplicate bug task BugD (35.7%). This task was by far the most difficult for participants to identify, and many participants reported that the word clouds generated by this task were not descriptive.

Unsurprisingly, participants frequently confused the word clouds generated for the app market research task PrMR and for the recommend tool task PrRec. These word clouds tended to have very similar key words, as both full task descriptions mentioned the same three productivity tools.

Comparing the results of the word clouds generated from segments to the ones generated from groupings did not reveal a substantial difference. This is a promising result, as it indicates that enough data can be generated from within the bounds of most task segments to create word clouds that accurately represent the topic of a task as a whole.

3.6 Discussion

Decisions we have made in designing the approach we introduce are impacted by the evaluations we undertook. We discuss threats to the validity of these evaluations and consider alternatives that could make it easier to apply our approach.
3.6.1 Threats to Validity

The evaluations of the approach we conducted rely on a data set that focused on six tasks. Although we chose these tasks to be examples of information finding tasks performed by developers, the range of tasks explored is small. By focusing on information finding tasks, we also exclude a significant category of tasks on which developers commonly work, namely coding related tasks. We believe that with minor adaptions, such as tokenizing camel case words or parsing the OCR results to extract in code comments, our approach could be made to work with coding tasks. If we took this approach to coding tasks, the quality of the code base in terms of documentation, naming convention, and so on, could play a large role in the ability of our approach to make accurate predictions. It would be impossible to associate a developer description, or generate a meaningful visual representation, if the code base does not contain descriptive names and lacks documentation. We leave the investigation of the generalizability of our approach across a wider range of tasks to future study.

Another threat to the findings is the size of the tasks studied and the interleaving
of work on different tasks. To fit within a reasonable time frame for a lab setting, the tasks worked on were relatively small in scope. In reality, developers work on complex tasks that can have a huge scope and span multiple topics. In addition, although we caused developers to switch tasks, it is not possible to replicate the many task switches a developer undertakes as he works [44]. A field study is likely needed to mitigate these threats.

We also note that the tasks we designed may be more specific in their wording than those that might occur in a developer’s normal work pattern. For example, a developer might work on a task in response to some relatively vague verbal request for help from a colleague. In such cases, it is unlikely that the summaries that the developer would write for these tasks are highly descriptive. We mitigate this threat by including a wide variety of low and high quality task summaries written by a group of MT workers with diverse demographic in our evaluation.

3.6.2 Applying the Approach

The approach we have introduced and evaluated assumes that the boundaries of a task segment can be known with high accuracy. Automatic detection of task segments (i.e., task switches) is a difficult problem (e.g., [48, 74]). We investigate this problem further in Chapter 4. While our own results are promising, and we are optimistic that the techniques to detect task switches will continue to improve, future work should explore the performance of our approach in the absence of knowing task segment boundaries. It may be that missing or erroneously predicting a task switch could lead to degraded performance in our approach in practice.

It is also possible that in practice the vocabulary a developer uses to describe their task does not match exactly with the words commonly found within the content of the task. For example, a developer might use the word “chart” in their task description, yet in the window content of the task the word “graph” might appear prominently instead. Applications of TF-IDF would miss this connection given its focus on exact word matches. Incorporating some notion of semantic similarity into our approach, for example adding Word2Vec or another model for word embedding, we might be able to enhance task descriptions to also include semantically similar words. More experimentation in a more realistic setting is
needed to investigate the impact and need for semantic similarity.

3.6.3 Artifact Access

Using OCR and capturing a developer’s screen content has several benefits. First, it is an application agnostic approach that does not require any instrumentation of applications. As well, a screenshot shows us the exact content a developer is looking at in the moment. While OCR performed well for the purposes of our analysis, there are many drawbacks that could limit its usability in practice. For one, OCR is an extremely CPU intensive task. Processing screenshots in real time in the background while a developer works may be impractical for this reason. An obvious alternative might be to send screenshots to the cloud for processing, but privacy concerns, both from the developer’s and company’s perspective, limit the applicability of this approach.

Another issue is the noise generated when using OCR. This may be alleviated to some extent by using a commercial option rather than the open source Tesseract engine. However, we can not guarantee that the product of a screenshot processed with OCR is exactly the same as the content a developer saw on their screen when the screenshot was taken.

An alternative which we will investigate in future work is to track all file accesses and edits made within the scope of a task segment. If we know which files a developer is interacting with, we can extract the contents of the file directly. The benefit of knowing exactly which information in a document is being viewed would be lost in such an approach. However, this loss may be outweighed by the ability to produce cleaner data, and the much lower CPU usage. The contents of web page visits could also be extracted relatively easily with the help of a browser extension. However, it could be difficult to obtain information from applications such as instant messaging and email clients, as there is a much wider range of choices for a developer to use in these cases. For this reason, producing a suite of instrumentations for all the most commonly used applications is impractical. Further investigation is needed to determine how much predictive power is lost by the exclusion of these categories of applications.
3.6.4 Creating Vectors from Window Titles

As mentioned in Chapter 3.1 in addition to recording screenshots of a developer’s active window, our tool also recorded the window title of every window the developer accessed. Unfortunately, due to a recording error window title data was lost for 5 of the 17 developers in our data collection session.

To investigate whether the easier to collect information about application window titles might suffice for supporting our approach, we evaluated RQ1 with the window title data from the 12 developers, in place of the information extracted using OCR. Comparing the results of this evaluation with the results from the same 12 developers using screen content, we found that while the results were lower overall, the difference was modest (64.4% accuracy using window titles vs 70.3% accuracy using screen content). Figure 3.8 illustrates the differences in performance seen on a per task level. While screen content is a superior choice of data source in almost all cases, window titles seem like a viable alternative especially given the savings in CPU resources. Worth investigating is whether a combination of our approach using window titles and the other data extraction techniques mentioned above can rival the results we achieved using screen content.
Figure 3.8: Comparison of performance per task with window titles as the data source vs screen content. For both approaches, results are calculated across the 12 developers with window title data available.
Chapter 4

Detecting Task Switches and Their Types

In the previous chapter, we showed that it is possible to determine and describe the content of a task, making the assumption that the times at which task switches occur are known. In this chapter, we investigate the feasibility of automatically predicting these task switches, as well as predicting at a coarse grain, and with minimal data, the type of task being performed. We consider the following two research questions:

**RQ3:** Can we automatically detect the times at which a developer switches between tasks?

**RQ4:** Can we automatically determine the types of the tasks that a developer works on?

Previous researchers have proposed approaches to automatically detect switches between tasks, varying mainly in the features used (e.g., user input or application based), and the method applied (e.g., supervised versus unsupervised machine learning) [48, 74, 76]. Yet, the evaluations performed to study these approaches are often fairly limited in terms of the tasks and number of participants, and the results show that it is very challenging to achieve high prediction accuracy of task switches without too many false positives [48, 51, 81], or that one has to accept a high deviation in time of 3 to 5 minutes between predicted and actual task switches [73, 74, 76]. In addition, existing approaches in the software engineering
domain for detecting task switches are limited to the IDE and therefore do not capture non-development work, which can account for 39% up to 91% of the time developers spend at work [3, 21, 44, 61, 77].

In addressing RQ3, we investigate whether we can automatically detect task switches of professional software developers in the field with high accuracy, based on temporal and semantic features as extracted from their computer interaction inside and outside the IDE.

We were also interested in classifying the type of task a developer is working on, since the better we understand the context of a task, the better we can support developers. To the best of our knowledge, there has been only one approach so far that looked at the automatic classification of developers’ activities on a task level [5]. Yet, their examination was limited to specific development activities only, and did not consider the whole range of non-development tasks that developers are working on, such as administrative or planning tasks.

In addressing RQ4, we investigate the task types that software developers are working on more holistically, and explore how accurately we can predict them in the field.

To address our research questions, we performed two field studies: one with 12 professional developers in which we observed their work over a 4-hour period and logged the task switches and types without interrupting their work; and one with 13 professional developers in which we regularly prompted participants to self-report their task switches and types over a period of about 4 workdays and conducted a post-study questionnaire. By varying the study methods, we wanted to achieve a higher generalizability of our results and ensure that we take into account the effects of self-reporting while also capturing the breadth of developers’ tasks over multiple days. For both field studies, we also collected the participants’ computer interaction using a monitoring tool that we installed on their machine and that was running in the background. From the computer interaction data, we extracted a total of 109 temporal and semantic features. Our analysis of the data shows that we can use the automatically logged computer interaction data to train machine learning classifiers and predict task switches with a high accuracy of 87%, and within a short time window of less than 1.6 minutes of the actual task switch. Our analysis further shows that we are able to predict task types with an accuracy of 61%, yet that this
accuracy varies by task type. The features based on mouse and keyboard interaction generally hold the highest predictive power, while the lexical features we extracted from the application names and window titles have the least predict power in our approach.

4.1 Study Design

To investigate the use of computer interaction data for predicting task switches and types, we conducted two field studies, a 4-hour observational study and a multi-day study with experience sampling, with a total of 31 professional software developers initially. The observations and self-reports served as the ground truth of participants’ task switches and types, while we additionally gathered computer interaction data to extract features for our predictions. In both studies, we used the same definitions of tasks, task switches and types which we also shared with the participants. A brief overview of our study design is presented in Figure 4.1.

Figure 4.1: Overview of the study design and outcomes.
4.1.1 Study 1 – Observations

In our first study, we observed the work of 12 participants over a period of 4 hours to gather a richer understanding of developers’ task switches and types they work on.

Procedure While conducting the study, the observer followed a detailed protocol that we developed before the study. The first observation session was performed by both observers at the same time. A cross-check of the two observation logs showed an inter-rater agreement of 97%, suggesting a high overlap of observing the same tasks and task switches.

Before each observation session, the observer explained the study purpose and process to the participants and asked them to sign a consent form, to install a monitoring tool that tracks participants’ computer interaction, and to describe the tasks they were planning to work on during the observation. The observer also introduced herself to nearby colleagues and asked them to ignore her as much as possible, and collaborate with the observed participant as they would normally do. After that, the observer placed herself behind the participant to prevent distractions, while still being able to see the screen contents on the participant’s computer. Finally, the observer started the actual observation session and asked the participant to continue their work as usual.

We observed participants for a total of four hours each on a single workday: two hours before and two after lunch. For the observations, we followed Mintzberg’s protocol of a structured observation session [47]. The observer wrote in an observation log each time the participant switched from one task to another. Each entry in the observation log consists of a timestamp, a description of the reason for the task switch and a description of the task itself. We inferred tasks and their details from the active programs and their contents on the screen, as well as discussions participants had with co-workers. After each session, the observer validated the observed tasks and task switches with the participant, by going through the list of observed tasks and accompanying notes and modifying mistakes made during the observation.

1 We used our own observation logging tool: https://github.com/casaout/ObservationStudyTool
**Participants**  We recruited 14 participants through professional and personal contacts from two large-sized and one medium-sized software companies. We excluded two participants for which we were not able to observe a sufficient amount of task switches (less than 10). Of the remaining 12 participants, 1 was female and 11 were male. Throughout this chapter, we refer to these participants as P1 to P12. Our participants had an average of 10.8 (±7.4, ranging from 1 to 20) years of professional software development experience and were working in different roles: 8 participants identified themselves as individual contributors and 4 as developers in a leading position. All participants resided either in Canada or the United States.

**Monitoring Tool** To collect computer interaction data from developers, we developed and used our own monitoring tool, PersonalAnalytics\(^2\) for the Windows operating system. The tool tracks participants’ mouse and keyboard interaction, as well as their application usage. For the mouse, the tool tracks the clicks (coordinates and button), the movement (coordinates and moved distance in pixels), and the scrolling (coordinates and scrolled distance in pixels) along with the corresponding time-stamp. For the keyboard, the tool records the type of each keystroke (regular, navigating, or backspace/delete key) along with the corresponding time-stamp. For privacy reasons, we did not record specific keystrokes. Our tool further records the currently active application, along with the process name, window title, and time-stamp whenever the window title changed or the user switched to another application.

**Task Type Inference** We inferred task type categories by performing a Thematic Analysis\(^8\) on the basis of related work and our observation logs. The analysis process included first familiarizing ourselves with the observed task switches, open coding the observed and participant-validated tasks and accompanying notes, identifying themes, and categorizing the resulting themes into higher level task types. This process resulted in nine task type categories: Development, Personal, Awareness & team, Administrative, Planned meeting, Unplanned meeting, Planning, Other and Study. The task types are described in more detail in Table 4.4 and discussed in [42].

\(^2\)https://github.com/sealuzh/PersonalAnalytics Details can be found in [42].
Chapter 4.4.1. In contrast to a task (and the task type), an activity describes an event or happening that does not necessarily need to have a particular purpose (or task). For example, the activity *Web Browsing*, could be grouped into several task types, such as *Development* when the developer is reading API documentation online and *Planned meeting* when the developer is using an online-conferencing tool.

4.1.2 Study 2 – Self-Reports

To capture a longer time period and more breadth in developers’ work, we conducted a second field study with 13 participants over a period of 4 workdays each. For this study, we used experience sampling, in particular we regularly prompted participants to self-report task switches and types. By using experience sampling, we also wanted to mitigate the risk of a bias in participants’ behavior due to an observer sitting behind them, which, for example, could lead to participants being less likely to browse work unrelated websites.

*Procedure* Before the study, we emailed participants a document explaining the study goal and high-level procedure, asked them to sign a consent form and to answer a pre-study questionnaire with questions on demographics, their definition of a task, reasons for switching between tasks, and on the task types they are usually working on. Afterwards, participants received the study instructions, detailing the study goals, definitions of task switches and types that we used for the study, and instructions on how to install and run the monitoring tool. They were asked to install the same monitoring tool that we described above on their main computer. In case participants worked on multiple computers (e.g., a desktop and a laptop), we asked them to install the monitoring tool on both devices. Participants were further asked to read our definitions of a task, task switch and task type, as well as instructions on how to use the self-reporting component that we added to our monitoring tool. Finally, participants were asked to pursue their work as usual for the next couple of workdays while also self-reporting their task switches and types when the pop-ups/prompts appeared.

For this study, our tool prompted participants once per hour to self-report their task switches and types for the previous hour. The self-reporting step is explained
in more detail below. We intentionally decided to use an interval of one hour rather than a full day, to balance the intrusiveness of the prompts with the ability to accurately remember tasks and task switches over the previous time interval [84]. To further ensure high quality in the collected self-report data we further allowed participants to withdraw from the study at any point in time, and to pick the time for their participation themselves. In addition, and to avoid boredom or fatigue, we asked participants to respond to a total of 12 to 15 prompts, assuming an average of four self-reports per day and a total of three to four workdays for participation. This number was a result of several test-runs over multiple weeks and from qualitative feedback gathered with a pilot participant, a professional developer. Furthermore, we provided support to postpone self-report prompts for 5 minutes, 15 minutes, or 6 hours, and built and refined the self-reporting component to require as little effort as possible to answer, e.g., by letting participants answer the required fields by simply clicking on elements instead of asking them for textual input. Finally, each pop-up also asked participants to report their confidence with their self-reports.

Throughout the study, participants could check the number of completed pop-ups. Once they completed 12 pop-ups, participants could notify us and upload the collected data and self-reports to our server. The upload wizard once again described the data collected and allowed participants to obfuscate the data before sharing it with us. At the end of the study, participants were asked to answer a post-study questionnaire with questions on the experienced difficulties when self-reporting task switches and task types, on further task types they were working on, and on how they could imagine using information on task switches and types. After completing the survey, participants were given two 10 US$ meal cards to compensate for their efforts.

Participants We recruited 17 participants through professional and personal contacts from one large-sized software company. We discarded data from three participants that self-reported less than 10 task switches in the days of their participation. We further discarded the data of one participant whose definition of a task switch was very different to ours and the rest of the participants (i.e., he considered every application switch a task switch). Of the remaining 13 participants that we used for the analysis, 2 were female and 11 were male. Our participants had an average
of 12.1 (±8.2, ranging from 1 to 30) years of professional software development experience and were working in different roles: 10 identified themselves as individual contributors and 3 as developers in a leading position (i.e., Lead or Manager). All participants resided in the United States. In this chapter, we refer to these participants as P13 to P25.

**Self-Reporting Component**  The self-reporting component is part of our monitoring tool and includes a pop-up with three pages. The first page asked participants to self-report the task switches they experienced in the past hour. It visualized participants’ application usage on a timeline using different colors for each application and allowed them to self-report their task switches by clicking on the lines denoting applications switches. We restricted the task switch self-reports to a granularity of application switches with a minimum length of 10 seconds for a variety of reasons: First, we assumed that most of participants’ task switches coincide with application switches (e.g., switching from the email client to the IDE, or from the browser to an IM client) and fewer happen during a session uniquely spent within the same application (e.g., switching tasks directly in the IDE or in the browser). And, we wanted to avoid cluttering the user interface of our self-reporting component and simplify the reporting for participants. Similar to [76], the timeline visualization provided additional details when the participant hovers over an application, such as the application name, time when it was used, window title(s) and user input produced in that application. As soon as participants completed self-reporting their task switches for the whole previous hour, they could proceed to the second page
and self-report their task types (see Figure 4.2). On the second page, we visualized the same timeline as before, but added another row that prompted participants to select task types from a drop-down menu. After selecting the task types for all task segments, participants could proceed to the last page. The third page asked participants to self-report their confidence with their self-reports of task switches and task types on a 5-point Likert-scale (5: very confident, 1: not at all confident) and optionally add a comment. Capturing participants’ confidence served as an indicator of the quality and accuracy of their self-reports. The user interface we used to collect the ground truth for task switches and types resembles the one by Mirza et al. [48–50].

The supplementary material [45] includes the study instructions we shared with participants, the pre- and post-study questionnaires they answered and additional screenshots detailing the self-reporting component.

### 4.2 Data and Analysis

For this study we collected two rich data sets, including observed or self-reported ground truth data, and automatically tracked computer interaction data. Prior to the main analysis of the data we performed multiple pre-processing steps, including data segmentation and feature extraction, which are summarized in the remainder of this section.

#### 4.2.1 Collected Data

For study 1, we collected observation logs for a total of 51.7 hours of work and an average of 4.3 (±1.3) hours per participant. For study 2, we collected self-reports for a total of 58 workdays and an average of 4.5 (±1.7) days per person. On average, participants reported a high confidence with their self-reports (>3) in 20.6 (±9.0), and a medium or low confidence (≤3) in 22.2 (±16.7) of the pop-ups they answered. 77.0% was the highest ratio of medium or low confidence self-reports that one participant had, and 16.7% was the lowest. We decided to only use the data of the 268 self-reports with a high confidence (>3), thus including a total of 268 hours of work and discarding the rest (289 self-reports). This allowed us to ensure we were training our models with data that is of high quality and accuracy. Future work
could also account for over- or under-confidence in participants’ self-reports.

Table 4.1 reports statistics on the self-reports. Since overall, only 11% of the pop-ups were postponed by participants, one reason for the relatively high number of self-reports with medium or low confidence could be that the pop-ups appeared at inopportune moments and participants did not remember they could postpone it. Instead, participants might have just clicked through the pop-up and reported a low confidence to not distort the data.

Table 4.1: Self-reports for Study 2.

<table>
<thead>
<tr>
<th></th>
<th>All</th>
<th>per Part.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Days participated</td>
<td>58</td>
<td>4.5 (±1.7)</td>
</tr>
<tr>
<td>Pop-ups displayed to participants</td>
<td>557</td>
<td>42.8 (±21.6)</td>
</tr>
<tr>
<td>Pop-ups answered by participants</td>
<td>268</td>
<td>20.6 (±9.0)</td>
</tr>
<tr>
<td>Pop-ups answered within 5 minutes</td>
<td>158</td>
<td>12.2 (±6.3)</td>
</tr>
<tr>
<td>Pop-ups answered after 5 minutes</td>
<td>110</td>
<td>8.5 (±5.4)</td>
</tr>
<tr>
<td>Pop-ups postponed by participants</td>
<td>62</td>
<td>4.8 (±3.5)</td>
</tr>
<tr>
<td>Pop-ups discarded by researchers</td>
<td>289</td>
<td>22.2 (±16.7)</td>
</tr>
</tbody>
</table>

4.2.2 Time Window Segmentation

To calculate and extract task switch detection features, we defined the time windows to be between two application switches, which we call application segments. Thus, the task switch detection model that we were going to build, could recognize task switches whenever a developer switches between applications, but would miss task switches within an application, such as a switch from a work item to the next one inside the IDE. We consider application segments to be an appropriate time window with minimal prediction delay, since developers spend on average only 1 to 2 minutes in an application before switching to another [22, 43, 44], and to ensure the accuracy of participants’ self-reports (in study 2) was high. Threats to this classification are discussed in Chapter 4.6. In contrast, previous approaches predominantly used longer and fixed window lengths of 5 or 10 minutes [51, 55, 74, 76]. These shorter and more flexible time windows at borders of application switches allow to more accurately capture developers’ behaviors, and to more precisely locate the point in time of the task switch. For the task type detection features, we used the time windows between two task switches, as identified by our observations (study 1) or participants’ self-reports (study 2), which we call task segments for the
A next step towards building a classifier for task switch detection is to extract meaningful features from the raw computer interaction data collected by the monitoring tool. The features that we developed are either based on heuristics that participants stated as indicative of their task switches in the post-study questionnaire (study 2), based on features that have been linked to developers’ task switching behavior in prior work, as well as based on our own heuristics. The features we used are presented in Table 4.2 and are discussed in more detail in the remainder of this section.

Task switch detection is a special case of change-point detection [6,24], which is the process of trying to detect abrupt changes in time-series data. This is why many of our features compare the similarity between characteristics of the previous application segments with the current one, for example the difference in the number of keystrokes. To determine how many steps back one needs to compare the current with the previous application segments’ features, we run the task switch detection taking into account 1 and up to 10 steps back into the past, and comparing the resulting precision and recall. Our analysis of the results indicated that after an initial increase of the precision for detecting a switch, the precision and recall gradually drop as the number of steps increases. We therefore chose 2 as the number of steps to go back in terms of application segments. As a result, the total number of features used for the task switch detection is 84, which is double the number of unique features used: once calculated for comparing the current with the previous application segment, and once to compare the previous two application segments. In the following, we provide an overview over all the features used:

**User Input Features**  The first feature group are user input features. They are based on keyboard and mouse interaction, such as the difference in the number of keystrokes the participant pressed per second between this and the previous time window segment.
Table 4.2: Features analyzed in our study and their importance for predicting task switches and task types.

<table>
<thead>
<tr>
<th>Features</th>
<th>Import. Switch</th>
<th>Import. Type All</th>
<th>Import. Type UI</th>
</tr>
</thead>
<tbody>
<tr>
<td>User Input Features</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Keystroke differences (4): difference in the number of navigate/backspace/normal/total keystrokes pressed per second between the previous and current application/task segment [5,27,33,50]</td>
<td>16.4%</td>
<td>19.1%</td>
<td>29.9%</td>
</tr>
<tr>
<td>Mouse click differences (4): difference in the number of left/right/other/total mouse clicks per second between the previous and current application/task segment [5,27,33]</td>
<td>17.9%</td>
<td>13.9%</td>
<td>29.4%</td>
</tr>
<tr>
<td>Mouse moved distance (1): total moved distance (in pixels) of the mouse per second [27]</td>
<td>6.8%</td>
<td>8.7%</td>
<td>13.5%</td>
</tr>
<tr>
<td>Mouse scrolled distance (1): total scrolled distance (in pixels) of the mouse per second [5,33]</td>
<td>4.7%</td>
<td>5.0%</td>
<td>8.2%</td>
</tr>
<tr>
<td>Application Category Features</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Switch to/from specific application category (26): switch to/from a specific application category (e.g., messaging), while the previous one was different. Application categories considered: CodeReview [PS], DeveloperTool [44], IDE [44], Idle [PS],[37], IM [PS],[28], Mail [PS],[28], Music [PS],[5,28,48], Navigate, Read/Write Document [44], TestingTool [44], Utility, WebBrowser [PS],[44], Unknown</td>
<td>28.0%</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>Same application category (1): the current application category is the same as the one in the previous application segment, e.g., both are messaging [28,48]</td>
<td>1.4%</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>Time spent per application category (13): the percentage of the total duration of the task segment that was spent in each of the 13 application categories [33,50,76]</td>
<td>NA</td>
<td>34.3%</td>
<td>NA</td>
</tr>
<tr>
<td>Switching Frequency Features</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Difference in the window switches frequency (1): difference of the number of switches between windows of the same or a different application per second between the current and the previous application/task segment [33,55,76]</td>
<td>7.2%</td>
<td>13.7%</td>
<td>19.0%</td>
</tr>
<tr>
<td>Difference in the time spent in an application (1): difference of the total duration spent between the current and the previous application segment [76]</td>
<td>9.4%</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>Lexical Features</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Code in window title (1): the window titles of the current and previous application/task segments both contain code, as identified by text that is written in camelCase or snake_case. Can also distinguish between development and other file types</td>
<td>1.4%</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td>Lexical similarity of the window titles and application names (2): cosine similarity based on the term frequency-inverse document frequency (TF-IDF) between the current and previous application segments’ window titles or application names [9,55,76]</td>
<td>6.8%</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

References on these features (in blue) are either on previous related work or participants’ suggestions (PS). A feature importance of NA denotes that the feature was not used for the prediction group. For the task type columns, ‘All’ denotes that all features were considered, ‘UI’ indicates that only the user interaction features were used, and the application category features were ignored. Numbers in brackets show feature counts.
Application Category Features  We categorized commonly used applications into one of 13 predefined application categories, based on our classification in previous work [44] and participants’ suggestions of what they consider to be good indicators for switching to another task. These include categories specific to software engineering, such as DeveloperTool, CodeReview or TestingTool, but also more general ones, such as Read/Write Document, Email and Web Browser. They are leveraged in 26 features that capture switches to or from a specific application category, such as switching to a messaging application or becoming idle. Since switching to another application might be another indication for a task switch [28, 48], we added one feature that captures these.

Switching Frequency Features  In the post-study questionnaire, participants mentioned that they often navigate through several applications to clean-up their computer right before starting a new task, which is why we added a temporal feature based on the window switching frequency. One feature captures the difference in the time spent in an application, since this might be another indicator for a task switch, either because a switch is less likely immediately after a task switch, and the likelihood of a task switch increases as time passes [76].

Lexical Features  Inspired by prior work [9, 55, 76], we also added three lexical/semantic features that are extracted from application names and their window titles. The textual data was first pre-processed to produce lists of words via tokenization on punctuation and whitespace. From these lists we also removed common stop words such as “and”, “the”, and “or”. Since window titles might include code snippets, such as a class or method name or development file type, we added a feature that captures whether the window title contains text written in camelCase or underscore_case, and whether this is different to the previous segment. To determine whether the previous and current application segments have a contextual similarity, two features are calculated based on the cosine similarity of the window titles and application names using the term frequency-inverse document frequency (TF-IDF). Note that the application name and window titles were also used to determine the application category features. In addition, and unlike some previous
work, we explicitly did not capture file contents to reduce intrusiveness and avoid privacy concerns [65, 80, 88].

4.2.4 Task Type Features
For the task type detection, we reused the same features as in the task switch detection whenever possible. However, some features required adaption or made no sense in this context. First, as the time window for task type detection encompasses one or multiple application segments, we replaced the application category features with a feature that captures the ratio between the time spent in the specific application category and the time spent in the task segment. This allowed us to determine the dominant application category in a task segment. Second, we eliminated the lexical similarity features as these are computed based on an application segment’s similarity to another segment. In the task type detection scenario, we have no comparable ground truth to use to calculate such features. This resulted in a total of 25 features used for the task type detection.

4.2.5 Outcome Measures
For the task switch detection, we labeled each application segment either with Switch or NoSwitch, depending on whether we observed a task switch (in study 1) or whether the participant self-reported a task switch (in study 2). While our model is able to detect task switches on the granularity of application segments, an actual switch might happen while using the same application. Thus, our task switch detection approach is at most the duration of the application segment away from the actual task switch, which was an average of 1.6 minutes (±2.2) in our study. For the task type detection, we labeled each task segment with the observed or self-reported task type. Descriptive statistics regarding participants’ task switching behavior and the task types they worked on can be found in Chapter 4.3.1 and Chapter 4.4.2, respectively.

4.2.6 Machine Learning Approach
We used scikit-learn [59], a widely used machine learning library for Python, to predict task switches and task types. We evaluated several classifiers by applying them
to our feature set and testing different hyperparameters. A RandomForest classifier with 500 estimators outperformed all other approaches, including a Gradient Boost-Classifier, Support Vector Machine (SVM), Neural Network and Hidden Naïve Bayes classifier. Details on the hyperparameters of the evaluated classifiers can be found in the supplementary material [45]. A RandomForest classifier is one form of ensemble learning that creates multiple decision tree classifiers and aggregates their predictions using a voting mechanism [10, 35]. It does not require a pre-selection of features and can handle a large feature space that also contains correlated features. Hence, for the remainder of this chapter, the presented results were obtained using a RandomForest classifier. Prior to classification, we impute missing values by replacing them with the mean and apply standardization of the features, which centers the data to 0 and scales the standard deviation to 1. These common steps in a machine learning pipeline can improve a classifier’s performance [60]. For the task switch detection, we further apply Lemaître’s implementation of SMOTE, which is a method for oversampling and can considerably boost a classifier’s performance in the case of an imbalanced dataset such as ours [34]. For the task type detection, where as much as 80-90% of the reported types are of the Development class we instead employ penalized classification to correct problems caused by class imbalance, as SMOTE has significant drawbacks when the minority classes have a limited number of samples [53].

We built both individual and general models, where an individual model is trained and tested with data solely from one participant and a general model is trained on data from all participants except one, and tested on the remaining one. Individual models often have a higher accuracy since they are trained on a person’s unique behavioral patterns. On the other hand, general models are usually less accurate but have the advantage of solving the cold-start-problem, which means that no prior training phase is required and the model can be applied to new users immediately.

To evaluate the individual models, we applied a 10-fold cross-validation approach, where the model was iteratively tested on 1/10 of the dataset while being trained on the remaining data. We adapted the cross-validation approach to account for the temporal dependency of the samples. In particular, there is a dependency between samples in close temporal proximity, since data from the preceding samples
is incorporated in the features. To ensure a valid and realistic evaluation of the model [63], we therefore deleted \( h \) samples on either side of the test set block. In our case, we chose \( h=10 \) since we included up to 10 preceding samples in the feature calculation (see Chapter 4.2.3). The cross-validation approach is illustrated in Figure 4.3.

![Cross-validation approach for the individual models, leaving a gap of 10 samples before and after the test set to account for the dependence of samples in close temporal proximity.](#)

**Figure 4.3:** Cross-validation approach for the individual models, leaving a gap of 10 samples before and after the test set to account for the dependence of samples in close temporal proximity.

### 4.3 Results: Detecting Task Switches

#### 4.3.1 Descriptive Statistics of the Dataset

Participants switched frequently between tasks, with a mean task switch rate of 6.0 (\( \pm 3.7, \min: 1.8, \max: 18.9 \)) times per hour. The average time spent on each task was 13.2 (\( \pm 7.3, \min: 3.1, \max: 30.8 \)) minutes \(^3\). Developers’ task switch behaviors are similar to previous work [22, 43].

#### 4.3.2 Task Switch Detection Error Distance and Accuracy

To analyze how well our task switch predictions work, we run a first discrete analysis by calculating the **error distance** between each predicted and the actual task switch. The average error distance is 2.79 (\( \pm 2.30 \)) application-switches, meaning that in

\(^3\)We do not report individual results for the two studies, since the task switch rate (\( p \)-value=.056) and time spent on a task (\( p \)-value=.215) are not significantly different in the two datasets.
case a task switch was not detected at the exact moment, it is on average 2.79 applications before or after the predicted one. To put this in context, multiplying the average application segment length of 1.6 (± 2.2) minutes (Chapter 4.2.5) with the error distance results in an average of only 4.46 minutes that a task switch is predicted before or after the actual one. Of all task switches that were not detected at the exact moment, 44.7% of the task switches our model predicted have an error distance of 1 application-switch, 15.8% have a distance of 2 application-switches, and 39.5% have a distance of 3 or more application-switches.

Table 4.3 gives an overview of the task switch detection performance of individual and general models. We split the presentation of the data into the two studies, since they were collected with a different method. As a baseline, we report the results of a random classifier, where the likelihood of predicting a certain class is based on the class distribution of the training set.

Overall, our analysis revealed that we can detect task switches at their exact location with a high averaged accuracy of 84% (precision: 62% and recall: 35%, kappa: 0.34) when trained with individual models. Applying the general model, we achieved an averaged accuracy of 73% as well as higher recall of 55% and lower scores in both precision (46%) and kappa (0.27). Overall, despite these differences we found the two models were similar in performance judged by both AUC (74% for individual vs 75% for general) and F1-score (43% vs 40%). Compared with our baseline classifier, both the individual and general model show substantial improvements across the board, with the exception of recall in the individual model. Note that this does not mean that the baseline necessarily performed better in this case, only that our model was more much selective in its predictions, as is reflected in the higher precision score. For the individual models, we compared the results of each participant’s model (see supplementary material [45]). It reveals that the prediction performance varies quite substantially for each participant.
<table>
<thead>
<tr>
<th>Dataset</th>
<th>INDIVIDUAL MODELS</th>
<th>GENERAL MODEL</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Accuracy</td>
<td>AUC</td>
</tr>
<tr>
<td>Study 1: Observations</td>
<td>82%</td>
<td>76%</td>
</tr>
<tr>
<td>Study 2: Self-Reports</td>
<td>86%</td>
<td>72%</td>
</tr>
<tr>
<td>All</td>
<td>84%</td>
<td>74%</td>
</tr>
<tr>
<td>Baseline</td>
<td>55%</td>
<td>49%</td>
</tr>
</tbody>
</table>

Table 4.3: Overview of the performance of detecting task switches, for both individual and general models.
4.3.3 Task Switch Feature Evaluation

A Random Forest classifier can deal well with a larger number of features which makes prior feature dimensionality reduction of our 84 features obsolete \cite{10,35}. While we do not apply a feature selection technique in our approach since it would only select the most predictive features in the model, we are still interested in learning if certain features are generally more important, especially across different participants. The second column of Table 4.2 contains the feature importance as attributed by the RandomForest classifier using all features and averaged over all participants’ individual models. To calculate the feature importance metrics, we used the Gini impurity measure from scikit-learn, which captures the feature’s ability to avoid mis-classification \cite{59}. The most predictive feature groups are user input (45.8%) and application category (29.4%). The feature group with the least predictive power are the lexical features (8.2%). The supplementary material includes the feature importances of each individual feature \cite{45}.

4.4 Results: Detecting Task Types

4.4.1 Identified Task Type Categories

As described in more detail in Chapter 4.1, we inferred task type categories after collecting task and task switch data from observing 12 developers at work and performing a Thematic Analysis. This resulted in nine task type categories we described in Table 4.4. In the post-study questionnaires of study 2, participants reported that they agreed with the identified task types and generally had no issues to assign them. However, two participants mentioned that a task type for Support duties was missing:

“[Support]-Duties. These are very specific tasks that require a lot of different things to do. It’s not Development and it can be a lot of ad-hoc and requires many context switches.” - P14

Two participants mentioned that it was sometimes difficult to know if time spent on emails should be assigned to Development or Awareness & team:
“I was sometimes unsure of how to classify the time I spent responding to emails. I generally classified it as development since most of the emails were development-related.”

- P21

Most of our task type categories are consistent with previous work that investigated knowledge workers’ tasks [15, 31, 33, 66]. For example, Meetings, Administrative, Planning and Private were also prevalent in both Kim et al.’s and Czerwinski et al.’s work [15, 31]. Kim et al. further divided project work (in our case Development tasks) into Documenting and Conceptualizing Ideas, Environment and Development and Design. We did not make these finer-granular distinctions since we did not want to make the self-reporting of task types in the second study too complicated, which would degrade the quality of self-reports.
Table 4.4: Overview and descriptions of the task type categories, the average time developers spent on each task type per hour of work, and the performance of our task type detection approach, for both individual and general models.

<table>
<thead>
<tr>
<th>Task Type Category</th>
<th>Avg (Stddev) mins/h</th>
<th>Sample Size</th>
<th>INDIVIDUAL MODELS</th>
<th>GENERAL MODEL</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>All Features</td>
<td>UI Features</td>
</tr>
<tr>
<td>Development</td>
<td>37.2 (±12.2)</td>
<td>612</td>
<td>70%</td>
<td>85%</td>
</tr>
<tr>
<td>Personal</td>
<td>9.7 (±7.0)</td>
<td>170</td>
<td>48%</td>
<td>45%</td>
</tr>
<tr>
<td>Awareness &amp; team</td>
<td>5.3 (±6.0)</td>
<td>234</td>
<td>64%</td>
<td>53%</td>
</tr>
<tr>
<td>Administrative</td>
<td>4.0 (±3.6)</td>
<td>12</td>
<td>50%</td>
<td>17%</td>
</tr>
<tr>
<td>Planned Meeting</td>
<td>3.6 (±2.7)</td>
<td>94</td>
<td>40%</td>
<td>40%</td>
</tr>
<tr>
<td>Unplanned Meeting</td>
<td>3.1 (±2.8)</td>
<td>90</td>
<td>43%</td>
<td>29%</td>
</tr>
<tr>
<td>Planning</td>
<td>3.0 (±3.5)</td>
<td>90</td>
<td>31%</td>
<td>24%</td>
</tr>
<tr>
<td>Other</td>
<td>2.7 (±3.9)</td>
<td>40</td>
<td>47%</td>
<td>25%</td>
</tr>
<tr>
<td>Study</td>
<td>1.9 (±1.9)</td>
<td>64</td>
<td>67%</td>
<td>58%</td>
</tr>
<tr>
<td>All</td>
<td></td>
<td>1406</td>
<td>59%</td>
<td>61%</td>
</tr>
<tr>
<td>Baseline</td>
<td></td>
<td>1406</td>
<td>30%</td>
<td>30%</td>
</tr>
</tbody>
</table>

The 'All Features' columns show results using models trained with all features, while the 'UI Features' columns show results from models trained using only user interaction features (i.e., excluding application category features).
4.4.2 Descriptive Statistics of the Dataset

On average, developers worked on 6.1 (±1.6, min: 3, max: 9) different task types during the studied time periods, indicating that most of the identified task types are relevant to all developers. The majority of developers worked on Development, Awareness & team, Personal and Planning tasks on a daily basis. Only five developers worked on Administrative tasks during the study period, indicating that for many developers this is not a task they spend time on very often. The task type participants self-reported having spent the most time on is Development, with an average of 37 (± 12) minutes spent for every hour of work. Participants also spent a surprisingly high amount of time, almost 10 minutes per hour of work, with Personal, including work unrelated browsing and messaging. Table 4.4 reports details for all task types as well as the number of participants who self-reported having worked on the task type.

We also analyzed if having a higher diversity in work (i.e., working on more different task types) correlates with developers switching more between tasks. There is a weak, not statistically significant positive correlation (Pearson’s $r = 0.32$, $p = 0.12$), which suggests that there are other, more important reasons causing developers to switch tasks.

4.4.3 Task Type Detection Accuracy

Table 4.4 shows the results of our task type detection approach across all 9 task type categories. We omit the accuracy metric in this table, as recall is a measure of individual class accuracy, and since the recall presented in the all row is weighted by class size it therefore assumes exactly the same value as accuracy. As with the task switch detection analysis, we trained both individual models and one general model which was trained on all participants. The Administrative task type was not predicted a single time by the general classifier, and as thus the precision scores were undefined for this class. Similarly, the task types Planning and Other had low precision and recall values, since the sample size used for training these types was small. In general, the individual models (precision 59%, recall 61%) outperformed the general model by a large margin (precision 44%, recall 50%).

One important aspect of our approach that distinguishes our classifier from
previous work (e.g., [5, 28, 48]) is its ability to make predictions even on previously unseen applications. To demonstrate this, we split the results into two categories: with the manual application category mappings (All Features) and without (UI Features). The UI Features include all user interaction features, but exclude application features. While the combined approach proved to be superior, user input features still proved to have high predictive power on their own. Overall, there was a 28.2% increase in precision when including the application category features, and a 24.5% increase in recall.

We also found there was a substantial difference in performance depending on the task type category. The Development task type proved to be the easiest to predict, achieving high recall (85%) and precision (70%) scores. Conversely, the Planning task type saw very poor results, with only 24% recall and 31% precision. These results are somewhat in line with what one might expect. Naturally, some task categories are more difficult to predict than others. For instance, discerning the nature of a meeting (planned or unplanned) based purely on a users applications

Figure 4.4: Confusion matrix for task type prediction.
used and input activity seems to be nearly impossible. As seen in Figure 4.4 there is substantial confusion between some categories, especially between the two meeting categories (Planned Meeting and Unplanned Meeting) and the Personal category. These categories tended to have a high amount of time spent idle, meaning the participant was away from their computer which naturally makes correct predictions exceptionally difficult. As a consequence of the dominance of Development samples in our dataset, our classifier also exhibits a strong bias towards predicting the Development category. While a larger sample size would likely help reduce this bias, it is of note that the Development category is also the one participants spent the majority of their time in, 37.2 (±12.2) minutes on average for every hour of work.

4.4.4 Task Type Feature Evaluation

The third and fourth column of Table 4.2 show the Gini feature importances we calculated for our RandomForest classifiers, averaged over all participants. When considering All Features, we found the time spent per application category features to have by far the greatest importance (39.1%), followed by keystroke features (17.5%). However, the combined user input feature group contributed more than any other feature group (47.6%). The lexical features did not contribute at all to the results of the classifier, which suggests there is room for improvement in this area as window titles can contain a substantial amount of hints that could help to identify a specific task. The supplementary material includes individual task type feature importances.

4.5 Discussion

In this section, we discuss implications of our results, possible improvements to automated task switch and type detection, and practical applications of automated task detection in real-world scenarios.

4.5.1 Improving Task Switch Detection

We found that for the task switch detection, the individual models perform quite similarly to the general model overall, even though the prediction performance varies quite substantially for each participant. This suggests that using the general
classifier is accurate enough to solve the cold-start problem. For practical, real-world applications, we therefore suggest using a general model as a default, and then allowing the user to improve the classifier by training it. As we found in study 2, collecting periodic self-reports over just a few days is feasible in real-world scenarios and may even lead to some insights about work itself.

More research is required to explore reasons for and better balance the individual differences in developers’ task switching behaviors. This includes investigating the characteristics of inaccurately classified task switches and consider additional data sources. For example, we could imagine to include information about a developer’s personality and company culture to train a classifier that works well for developers with similar work habits, instead of building a general one for everyone. Future work could also study the predictive power of features extracted from additional data sources, such as emails, calendars, biometrics (e.g., detecting when a user is away from the computer), and more detailed development related data (e.g., activities inside the IDE).

The relatively low feature importances of our lexical features shows further potential to more effectively leverage contextual information. Besides calculating lexical similarity based on cosine similarity (TF-IDF) of window titles, we also experimented with variations, such as an unweighted term frequency metric and two different word embedding models including one trained on Wikipedia, and one trained on StackOverflow which has been shown to produce embeddings that are more closely related to the domain of software engineering\[^{19}\]. They led to even less predictive features, which is why we did not report them separately. One reason could be the little overlap in the window title data. Window titles generally capture only the application name and the name of the current file, document, email or website, which limits overlaps with other window titles. Including the actual contents of these resources could be one way to overcome these limitations, but could result in privacy concerns, as discussed in more detail in the next section.

### 4.5.2 Improving Task Type Detection

For the task type predictions we found that the individual models outperform the general model, with an overall accuracy of 61% compared to 50%. Even though we
collected data from a rather large sample of 25 participants (compared to similar work), we were not yet able to build highly reliable general models, which could solve the cold-start problem. The difficulty to discover common patterns across all participants emphasizes how individual and diverse developers’ tasks are.

We see our work as a first step towards better understanding and automatically characterizing developers’ task context. With our models’ ability to automatically detect task switches based on data collected through our computer interaction monitoring, a next step could be to collect a more in-depth set of data in-between two task switches, and from more participants over a longer period of time. For example, IDE extensions (e.g. Feedbag [2] or WatchDog [7]) could be leveraged to identify the code files, code reviews, and projects the developer has been working on, browser trackers (e.g. RescueTime [67]) could identify and classify the websites a developer visited, and integrations into the email or IM client could help to understand which people a developer communicated with. To better manage these large amounts of data, research will need to come up with approaches to model and summarize task context—and task types are a first step into doing that. We have been able to find little work on automatically detecting, characterizing and summarizing (developers) tasks yet [5, 33, 50].

While more fine-grained lexical data, such as the file or website contents (as applied in [65, 80, 88]) or participants’ actual keyboard input, could be leveraged to improve our models, it also might reveal details about the company’s products or the developers’ work and personal life that they are not comfortable sharing. To minimize privacy concerns, we had to find a trade-off between intrusiveness, by capturing only a minimum set of data, and completeness, by monitoring as much as possible to get enough data that allowed us to predict task switches and types in the field. To earn participants’ trust with capturing potentially sensitive data, we were also transparent with what data we collect and how it will be used, allowed participants to review it before sharing it with us, and making it possible to pause the monitoring application at any time that seemed particularly sensitive to them.
4.5.3 Reducing the Prediction Delay

Ideally, a task switch and type detection would be close to real-time, i.e., close to the exact time a switch occurs. With our approach, there can be a prediction delay of a maximum of one unique application segment, on average 1.6 minutes (±2.2), when predicting a task switch. This delay is considerably smaller compared to previous approaches that applied fixed window lengths of (usually) 5 minutes (e.g., [33, 50, 51, 55, 74, 76]). Nonetheless, future work could further reduce the prediction delay by further shortening the smallest possible segment size, in our case application switches. This would allow to also identify switches within an application, such as when a developer is switching tasks inside the web browser or IDE.

4.5.4 Applications for Automated Task Detection

An active area of research aims to better support developers’ frequent task switching, for example by supporting resuming interrupted tasks or by easing task switching. So far, most approaches are limited to developers’ manual identification of task switches, and their evaluations have pointed out challenges this poses for them. Our approach demonstrates the feasibility of automatically detecting task switches and types in the field, based on a few hours of training data, which makes it possible to increase the value of previous approaches significantly and stimulate new research and tool support. Notably, tool support would greatly benefit from the improvements we discussed in the sections above. In the post-study questionnaire of study 2, participants described concrete applications that we qualitatively analyzed and related to prior work, which resulted in the following three main opportunities for applying automated task detection:

One application of an almost real-time detection of task switches that 8 (out of 13) participants described is to actively reduce task switching. This includes automatically blocking notifications from email, instant messaging or social networks when a developer is focused on a (challenging) task, to allow extended times of deep focus:
“What if Windows has a built-in and personalized model about when to give you notifications. I feel like there is a good middle ground between forcing the user to turn off notifications from the OS and having too many notifications interrupting the user.” - P25

Reducing task switching at times of high focus could greatly reduce multi-tasking, a major source of stress and quality issues [22, 38–40]. Similarly, an automated task switch detection could improve interruptibility classifiers and postpone in-person interruptions from co-workers to task switch borders, times they are less costly [20, 90, 91].

Another application of automated task detection could be to support the resumption of suspended or interrupted tasks. Participants did not suggest this application themselves, but 8 (out of 13) rated it as ‘useful’ or ‘very useful’ in a follow-up question of the final questionnaire. According to Parnin and Rugaber, a major challenge of task resumption is to rebuild the interrupted task’s context [57]. Applying similar summarization approaches as seen in other areas of software development [52, 85] could be presented to the user as cues upon returning to the suspended task, which has been shown to considerably reduce the resumption lag [1, 4, 71]. While previous approaches, such as TaskTracer [18], Scalable Fabric [68], GroupBar [78] and Mylyn [29], allow the capturing and presentation of task context, they require the user to manually group related artifacts or manually state the start and end of a task, thus, reducing chances of long-term adoption. Even tough there is room for improvement as discussed above, our approach can serve as a starting point to automate these approaches, since it can already be beneficial to receive help with resuming some tasks, as long as they are detected correctly.

A third opportunity of application that 10 (out of 13) participants suggested is to use automated task detection to increase their awareness about task work and time spent on tasks, which could help to identify opportunities for work habit and productivity improvements. This is in line with a survey with 379 developers that showed the most-often mentioned measurement of interest when reflecting about productivity are the tasks developers made progress on and completed in a workday [43]. An aggregated visualization of the automatically inferred tasks could give developers insights such as how much time they spend on different tasks, when they worked on planned versus unplanned tasks, or their multi-tasking behaviors:
“It can help point out different working styles that are also effective and efficient. Not everyone works in the same way.” - P24

Recently, researchers started building retrospective dashboards for developers [7, 12, 42, 86] and other knowledge workers [32, 67, 87], usually by visualizing data on the level of applications or application categories, but suggesting that a per-task level would be more beneficial. An increased awareness about one’s task switching behavior could support developers to identify goals that help to maintain and improve good work habits, such as reducing multi-tasking or actively blocking notifications from distracting services and websites at times they need to focus. Participants further suggested that the data could help to reduce administrative workloads that require them to report time spent at work:

“We’re often asked to report at the end of the month how much time we spent on support requests (...) versus development work. That kind of info is tedious to track manually, but a tool could generate an automatic report as needed, allowing for more accurate counts.” - P22

Lu et al. recently showed that the lack of logs of activities and tasks is often a hindrance to be able to transfer them into time reports [36]. While a few time-tracking tools already exist (e.g., DeskTime [17], TimeDoctor [83]), they all require users to manually specify the start and end of a task.

4.6 Threats to Validity

Observing Developers in Study 1 The internal validity of our results might be threatened by the presence of the observers during the observation sessions, causing developers to diverge from their regular work habits, e.g., having less breaks than usual. Observing participants on a single day only might not be representative of the participant’s regular workday. We tried to mitigate these risks by not interacting with participants during the observations, splitting up the session into two two-hour blocks, sitting as far away from the participant as possible, telling co-workers beforehand that they could still communicate and interrupt as usual, and by allowing the participant to pick an optimal timeslot that is representative of their usual work. Our observational study has the advantage that, rather than performing a lab
study or experimental exercise, participants were observed during their real-world work, thus increasing generalizability and realism. However, the above mentioned risks of observing developers at their workplace make it very difficult to scale observational studies and observe them over many days. Hence, we did not rely only on observations, but also on participants’ self-reports and with that, combining two methods and strengthening our overall approach.

Self-Reporting in Study 2 While collecting participants’ task data using self-reports has proven to be a valuable approach to scale the collection of labeled data for supervised learning, there are a few limitations. First, we rely on the accuracy of participants’ self-reports. For example, they might not always have been able to accurately remember their tasks, or filling out the pop-up regularly might be perceived as cumbersome after a while. In Chapter 4.1.2 we describe our actions to minimize these risks in detail, including the ability to postpone a pop-up and collecting confidence ratings. Aiming to make the self-reporting as easy as possible required limiting the self-reports to segments with the granularity of an application switch and excluding application switches shorter than 10 seconds. This is why our models are unable to detect task switches within an application, as well as very short ones. Since developers switch between applications very frequently, on average every 1.6 minutes (±2.2), our model is able to predict a task switch within the same time frame. Future work could investigate how to give participants good-enough cues that allow them to accurately self-report switches within applications (e.g., switching from a news website to the work item tracker in the browser) without making the interface too cluttered. Finally, the reliance on collecting computer interaction data only, instead of also including other sensors such as heart-rate monitors or cameras, limits our knowledge of what is happening when there is no input to the computer, e.g., in the case of idle times from using the smartphone, reading a document without scrolling, or a discussion with a co-worker.

Sample Size A further threat to the external validity of our results could be the number of participants. A higher number of participants might have led to a more robust general model to predict task switches and task types. Nonetheless, collecting
task data from 25 participants is considerably higher than what was reported in previous work (between 1 and 11 participants). We tried to mitigate this threat by selecting participants from four different software companies in various locations.

*Task Definitions* The construct validity of our results might be threatened by our definitions of a task (switch) and our open coding approach to identify task type categories. To minimize this risk, we based our definitions of task, task switch and task type on previous work, and asked participants about their own definitions in both studies (Chapter 4.1).
Chapter 5

Discussion

In the previous two chapters we presented two different approaches: the first to automatically identify and describe a software developer’s tasks, and the second to automatically detect a software developer’s task switches and types. But while we developed and evaluated these approaches in separate studies, they are highly complementary to each other. In this chapter we will discuss some of the ways in which the approaches could complement each other in future applications.

5.1 Improving Task Identification

In developing the task identification approach presented in Chapter 3, we made the assumption that a developer’s task switches were already known. In a real use case scenario, we cannot expect a developer to manually indicate all of their task switches. Such a requirement would limit the uptake of this approach, and even if developers intended to indicate their task switches, it is highly likely that they would forget to do so frequently [29]. By incorporating the approach presented in Chapter 4 for task switch and type detection, we could fully automate the process by automatically detecting the boundaries of tasks. In addition, the rough type categorization generated could be used to improve the generation of task representations. For example, certain task types might have some words which are strongly associated with them. If these words occur within the information extracted from a task, then they could be assigned a more prominent relevancy score.
5.2 Improving Task Switch Detection

The approach to task switch detection presented in Chapter 4 achieves reasonably high accuracy, precision and recall scores, especially in comparison to previous works. However, there is still substantial room for improvement. One direction for future work to investigate could be the use of a semi-automatic approach, in which task switches are predicted automatically but users are prompted occasionally to correct erroneously detected switches. Since it can be difficult for developers to remember what they worked on at a specific time during their work day, the task identification approach presented in Chapter 3 could be used to create word clouds to be used as visual aids to help a developer determine where the boundaries of a task correctly belong. Such a semi-automatic approach would also present an opportunity to gather further ground truth information. By adapting our task switch detection approach to use a different kind of model, for example a long short-term memory (LSTM) based model, this information could be used for online learning. This would allow for more individualized models to be made available to all users. LSTM models are also particularly effective at processing sequential data such as the data gathered from a developers computer interactions.

5.3 Fully Automatic Task Support

The approaches presented in Chapters 3 and 4 represent a step towards truly fully automatic task support systems. Revisiting the Mylyn system, the existing constraints which contribute to developer overhead when using the system are mainly in requiring that developers indicate their task switches, and in requiring developers to label their tasks. The former issue is resolved by our approach to task switch detection, while our approach to associating task descriptions and generating word clouds for tasks resolves the latter. Incorporating these approaches into an updated tool could mean developers would reap all the benefits of the Mylyn system without any of the overhead. Future research will be required to investigate the efficacy of such an update in comparison to the manual alternative, as well as developer opinions.
Chapter 6

Conclusion

Have you ever wondered what you worked on throughout a day, possibly to record time spent on different projects? Have you ever wanted to look back and find where you worked on a particular task to find what resources you consulted as part of the task?

This thesis introduces and evaluates approaches to help support these goals. First, an approach to automatic task identification which extracts the contents of the active window being worked with on a regular basis, uses optical character recognition (OCR) to transform the contents into tokens and words, and applies TF-IDF with work tokenization (NLTK) to form a vector representation of the task segment. For the purposes of this approach, task segments were formed using manually indicated task switches. The vector representation generated from this approach can be used to help identify which task a segment represents for a known set of tasks with an averaged accuracy of 70.6%. Visual representations of a task segment were also generated using TF-IDF scores for each word in a bag of words formed from the screen content of the task segment as a developer worked. Through a survey, we found that participants could determine which task a word cloud for a segment of work represented with reasonable accuracy (67.9% on average) for several tasks. Interestingly, the accuracy rose only modestly when considering identifying the task based on a word cloud formed from all segments comprising work on a task.

Second, an approach to automatic task switch and type detection that extends
previous work by using a broader range of temporal and semantic features, by
developing new features based on a developer’s computer interactions, and by
not being limited to capturing task switches and types within the IDE only. The
evaluation of this approach in a field-study with 25 professional developers, com-
pared to 1 to 11 participants in previous work, revealed higher accuracy (84% for
switches and 61% for types) and less delay in the predictions than comparable prior
work.

The results from the evaluations of these approaches show promise for helping
to determine automatically the intent and boundaries of a developer’s many tasks
throughout the day. By enabling the detection of developer intent within the context
of a specific task segment, various tools can be improved that a developer relies
upon and new tools can be introduced to help support such activities as time tracking
and task resumption.
Bibliography


Appendix A

Laboratory Data Collection

This appendix contains additional information pertaining to the data collection done in Chapter 3.

A.1 List of Tasks Performed By Participants

Duplicate Bug Task  Your manager has noticed that there has been a substantial influx of duplicate bug reports recently. Explore the provided list of bug reports, and identify whether there is a duplicate and provide its ID. search the bugs with the ids: 2264, 2268, 2271, 2277.

Viz Library Selection Task  Your team has developed an application for optimizing developer work patterns - reducing the number of impactful interruptions. You have been tasked with creating visualizations for a presentation outlining the benefits of your product to potential clients. You have the following data available to you for use: Application usage times, interruption times, durations, and disruptiveness levels, keyboard and mouse activity levels. What libraries would you suggest for creating data representations? Give some examples of other works that have been created using these libraries. At a minimum the visualizations should include a before/after comparison of developers work days using the product vs not using the product.
**App Market Research Task**  The software company you work for is considering expanding into the productivity tool sphere. Your manager has asked you to do some market research on 3 of the most popular already existing apps in this domain: Microsoft To-do, Wunderlist, and Todoist. Provide a short written summary of the similarities and differences between these 3 apps.

**Recommend Tool Task**  Your coworker is having difficulty deciding on which productivity app they should use, and have asked you for a recommendation. They have narrowed their decision to 3 apps: Microsoft To-do, Wunderlist, and Todoist. Based only on app store reviews, which of these apps would you recommend? Identify any reviews that were particularly influential in your decision.

**Deep Leaning Presentation Task**  You are preparing to give a presentation on potential deep learning applications to the CTO of your company. While you have already completed the slides for the presentation, you should also prepare answers for a few questions which are likely to arise during the presentation.

1. The lines drawn between layers of the network on the included slide represent weighted inputs from one layer to the next. How does the network decide on what weights to choose during the training process?

2. Most of the technologies behind deep learning have already been around for over 30 years. Why is deep learning only becoming popular now? What has changed?

3. What kind of performance increases can be seen by using GPU’s instead of CPU’s? Are GPU’s always superior with respect to deep learning applications?

**Blockchain Expert Task**  You recently gave a short presentation to your colleagues outlining different ways the company may be able to make use of blockchain. One of your co-workers felt a little bit lost during the presentation, and emailed you a couple follow up questions afterwards.
1. You mentioned that blockchain is a form of distributed ledger. What does this mean? What advantages are offered over traditional client-server database ledger systems?

2. Could you explain what ”proof-of-work” means? What is it? What does it do? Why is it necessary?

### A.2 Additional Figures

**Figure A.1:** An example screenshot captured with our tool of a developers inbox at the start of a data collection session.