Bachelor Thesis

January 31, 2018

Towards Automated Task Detection Based On User Interactions in IDEs

Nico Strebel

of Olten, Switzerland (14-705-602)

supervised by

Prof. Dr. Harald C. Gall

Sebastian Proksch

University of Zurich[UZH]

s.e.a.l.

software evolution & architecture lab

Bachelor Thesis

Towards Automated Task Detection Based On User Interactions in IDEs

Nico Strebel

University of Zurich[UZH]

s.e.a.l.
software evolution & architecture lab

Bachelor Thesis

Author: Nico Strebel, nico.strebel@uzh.ch

URL:

Project period: 01/10/2017 - 31/01/2018

Software Evolution & Architecture Lab
Department of Informatics, University of Zurich

Acknowledgements

I would first like to thank Prof. Dr. Harald Gall of the University of Zurich for giving me the opportunity to realize my bachelor's thesis at Software Evolution And Architecture Lab.
Especially, I would like to thank Sebastian Proksch who helped me throughout my thesis intensively. His door was always open and he would not spare any effort whenever I needed help. He guided me in the right direction and provided many detailed explanations for my questions. Additionally, without his prior research, this thesis would not have been possible at all.
I would also like to thank my family who has been supporting me throughout my life and studies. Special thanks go to my lovely sister, Corinne. She lectured my work and patiently drew my attention to logical gaps in my explanations, as well as a ton of missing commas.
Last but not least, I would like to thank the informatics student association of the University of Zurich, ICU. They not only provided me with a place to work but also helped me throughout my bachelor's studies.

Abstract

Integrated Development Environments are the standard tool used by programmers to develop software. Nowadays, IDEs record every action developers execute during their coding work. This thesis tried to make a step towards automated detection of task boundaries in order to allow reasoning about an event stream. Our approach uses machine learning techniques in order to find patterns which lead to task switches. For this purpose, we propose a layer above low-level events which depicts the task's state. Our evaluations suggest, that our approach is not completely mellow yet, but provides a good foundation for further research.

Zusammenfassung

Integrierte Entwicklungsumgebungen sind das Standardwerkzeug, welches Entwickler benutzen, um Software zu entwickeln. Entwicklungsumgebungen zeichnen heutzutage alle Aktionen auf, die Entwickler während ihrer Programmierarbeit ausführen. Diese Thesis hat versucht, einen Schritt in Richtung automatischer Erkennung von Aufgabengrenzen zu machen, um Schlussfolgerungen aus Entwickler Aktionen zu ermöglichen. Unser Ansatz benutzt maschinelles Lernen um Muster, welche zu Aufgabenwechsel führen, zu erkennen. Dafür schlagen wir eine Ebene oberhalb von primitiven Aktionen vor, welche den Zustand der Aufgabe veranschaulicht. Unsere Auswertungen haben ergeben, dass unser Ansatz noch nicht völlig ausgereift ist, jedoch eine gute Basis für weitere Forschungsarbeit bildet.

Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

Research on human behaviour has been conducted for centuries. For researchers, it is often not sufficient to know that people execute certain actions. They also want to understand why people behave in a certain way. Developers are no exception to this. So far however, research has been focused on analysing what developers do, but less so why.
Integrated Development Environments (IDEs) play a central part in the development of software systems. Understanding, how developers use IDEs, has gained a lot of interest over the last decade. Researchers have conducted countless studies based on interaction records generated from IDEs. However, these studies often fail to inspect the log data within its bigger picture. Developers conduct programming work based on their assigned development task. These tasks are usually not recorded in IDE interaction logs. Knowing which task developers are working on, allows for reasoning about the actions programmers execute. The goal of this thesis was to provide an approach to automatically detect task switches in low-level interaction logs and thus to gain context knowledge for developer interaction.

Development tasks usually require a lot of recurring actions such as building and test execution. This characteristic can be exploited by finding recurring patterns in low-level interaction records. For this purpose, an event data stream was converted into a higher-level form which portrays the task's state at any moment in time. A machine learning algorithm was then applied to find patterns in the state's attributes. These patterns lead to the identification of task switches.
Machine learning is able to model the relationship between a set of independent attributes and a dependant value. Such a model can predict task switches and thus provide valuable information for event logs.

The results obtained from our research suggest that solely based on interaction logs, it is not possible to build a globally applicable model for task boundary detection. However, we were able to prove that based on an event stream with contextual information, it is feasible to compute models that predict task switches for individual developers. The findings from this research provide following contribution to current task boundary detection research:

- A technique which makes low-level event logs suitable for supervised machine learning is provided.
- Based on enriched event streams, a sophisticated sampling tactic, and carefully chosen features, individual prediction models can be built.
- Due to the feasibility of automated task boundary prediction, it might be possible to gain more knowledge about developer behaviour from low-level in-IDE interaction logs without influencing the observed developers.

Chapter 2

Background

Automated task boundary detection in IDEs has not been the target of much research so far. However, we are convinced that extracting the task from interaction logs is necessary to understand why developers complete a task in a certain way. On the other hand, there has been a lot of research on task boundary detection for usual computer desktop work.
There is a lot of research from various computer science unrelated fields which provides invaluable insight on how to approach the problem of automated task boundary detection. This section provides an overview of the current state of in-IDE interaction tracking and how this problem was approached in other domains. It also lays out the foundation for our approach.

2.1 User Interaction Tracking

Tracking the behaviour of users has gained a lot of interest in the last twenty years. With the introduction of data mining and machine learning techniques, analysis of customer behaviour has risen significantly in importance and practicality [MFPSS96]. Researchers and application developers have spent a lot of effort to analyse and understand the behaviour of users in web applications. Findings from such analysis can be used to improve the user experience and customer satisfaction noticeably. Atterer *et al.* have conducted a case study on the benefits provided by user interaction tracking [AWS06]. Usability tests usually involve a lot of effort. Participants have to be found and often paid. Additionally, such tests often suffer from the well-known Hawthorne-effect [1]. Tracking the interactions remotely decreases these cost factors and additionally prevents the influence of users [AWS06].
Interaction tracking however is not only suitable for usability tests. Coenen *et al.* have introduced a framework for self-adaptive websites [CGS00]. They provide techniques, which allow web applications to dynamically alter their structure based on user interaction patterns.

Interaction Tracking in IDEs In the realm of software development, Integrated Development Environments (IDEs) have become the de facto standard tool for developing medium and large scale projects. IDEs bundle numerous applications which facilitate the software development process. Kersten and Murphy have developed *Mylar* (nowadays Mylyn), an extension for the Eclipse IDE which provides a task focused interface. *Mylar* was also one of the first tools that actively tracked developer interaction with IDEs [KM05]. *Mylar* uses the collected interaction data to support developers in focusing on coding tasks rather than code discovery. The tool was also

[1] Subjects change their behaviour when they are observed [Haw]

the foundation for further developer behaviour research. Murphy *et al.* have used data gathered by *Mylar Monitor* to analyse how developers are using the Eclipse IDE. They have shown that monitoring developers can prevent feature bloat by identifying rarely used IDE functionality. Additionally, interaction data provides valuable knowledge for improvements of IDEs and similar tools [MKF06].
Minelli *et al.* go even a step further. They have developed DFlow, a tool which tracks fine-grained interactions of developers within the IDE [MMLB14]. Minelli intends to use the data collected by DFlow to make progress towards self-adaptive IDEs [Min14].
Amann, Proksch and Nadi have developed FeedBaG, an interaction tracker for Microsoft's Visual Studio IDE [APN16]. FeedBaG records fine-grained developer actions within the IDE. The data collected by FeedBaG has been used for a large-scale study on how developers use Visual Studio [APNM16].

2.2 Task Boundary Detection

The data collected by user interaction tracking can be used for a wide variety of statistical analysis. Researchers have gained a lot of interest not only in detecting user behaviour patterns but even more so in understanding why users execute certain actions. Knowing the user's context and their reasons allows for a more sophisticated interpretation of logs. Therefore, it is crucial to know when a task starts and when it ends. Detecting switches between tasks automatically and remotely based on raw interaction logs facilitates this process perceptibly.
Franklin *et al.* have argued that understanding why users execute certain actions is fundamental to provide an intelligent user interface. They also show that knowing the user's task allows the system to cooperate without the need for explicit commands by the user [FBH02].
Shen *et al.* have developed a tool that associates files, mails, contacts and more with specified user tasks. Their implementation, *TaskPredictor*, consists of two systems that predict the user's current task. *TaskPredictor.WDS* analyses the focused window in order to identify the active task. *TaskPredictor.email* analyses incoming mails in order to detect task switches. They apply three different machine learning methods in order to assign a user task to the analysed point in time [SLDH06]. By treating task boundary detection as a supervised machine learning problem, they were able to correctly detect task switches in most cases.
Similarly, Oliver *et al.* [SOSt06] have developed SWISH, a tool which groups related windows that are active on the computer desktop. Based on the window title and the user's behaviour patterns when switching between tool windows, they were able to establish a method which can assign a window to its associated task.
Devaurs *et al.* [DRL12] have established an ontology-based approach for bundling simple events into abstract tasks in standard computer desktop work. They have proposed a bottom-up scheme according to which tasks can be identified from simple events. Whenever a user executes an action on his desktop, a low-level event is generated. Related events are subsequently aggregated into event blocks which consist of events that concern the same resource. Multiple event blocks are then combined in order to identify tasks.
Stumpf *et al.* [SBD+05] have used machine learning in order to automatically predict the task a knowledge worker performs. They also describe the major challenges in predicting task switches. Their research suggests that obtaining a high accuracy in predicting user tasks is impeded mainly by two characteristics: Tasks are often intertwined and thus, task boundaries are not clearly defined. Additionally, there are different ways to complete the same task.

Task Boundary Detection in IDEs Researchers have been successful in detecting task boundaries with different approaches. Most of them involve analysis of either the focused windows or the edited files. Since IDEs bundle multiple tools which are used in software development, they are the main window used by developers. Therefore, focused windows can not be exploited as much as in traditional task detection. Task boundary detection in IDEs consequently relies on the analysis of interactions within the IDE and the edited source files.
Coman and Silitti were able to provide a technique which can reliably divide development sessions into task related sections [CS08]. Their approach identifies the methods which are modified frequently during a session. They describe these methods as the core of a subsection. Additionally, their algorithm groups methods, whose modifications are overlapping, together. Their approach relies on the idea, that subsequent sections, that do not have the same core, indicate a task switch between them [CS08].
Coman's algorithm was intensively tested by Zou and Godfrey. Their case study suggests that the previously proposed algorithm has an error rate of 76% [ZG12]. They identify the discrepancy between their and Coleman's results in its evaluation setup. Coleman tested his technique in a laboratory environment which did not suffer from external disturbance. However, in industrial work, interruptions happen frequently. Coleman's algorithm fails to acknowledge them though. Therefore, Zou and Godfrey suggest that task boundary detection is in fact a two part problem: Identifying sessions and linking related sessions together.
Kevic and Fritz have shown that task boundaries can be detected accurately by analysing the developer's code modifications [KF17]. They used a regression model to distinct between unrelated and loosely related methods. Identifying these unrelated functions in source code allowed them to detect task changes with a reliability of over 80%. Their work concerned aiding developers during change tasks by suggesting related code elements based on recent activities. For this application, knowing when a task changes is crucial.
The two approaches explained above suggest that - similar to task boundary detection in web applications and office work - task switches can be detected by analysing the modified artefacts, i.e. source files and methods. However, these approaches require close tracking of the modified artefacts. Therefore, these techniques are not applicable to IDE interaction logs.

2.3 Task Management in Software Development

The scope of software projects can seldom be dealt with by a single developer. One of the main factors of project management concerns the break down of the work load. Modern development processes thus rely on effective task management. Sophisticated division of work packages allows a team of developers to work on the same project in parallel and thus decreases the time required for software projects.

Task Definition Since the publication of the Agile Manifesto [BBVB+01], agile methodologies have been taking over the way software projects are organised. Agile methods strive to allow for quick delivery and quick change. These techniques most often rely on iterative development processes [GH11]. Agile methods describe project requirements with so called "User Stories" [Coh05]. A User Story usually follows a well-defined template [2]. The workload created by a story is split up into *tasks*. A task should ideally be completable within a day at most [Coh05]. Additionally, tasks are supposed to be independent from each other and only concern related work. However, this is not always the case in practice.

[2] Example: As a <role>, I want to <desire>.

One of the central aspects when managing tasks is called Definition of Done (DoD). It is strongly advised that the DoD is clearly defined by the project team and understood by every member. Consequently, there is no global DoD but lots of individual ones.
Davis has provided an exemplary DoD [Dav13]. His example definition includes recurring actions such as peer-reviews and unit tests. DoDs not only apply to tasks, they also describe necessities for user stories and even complete projects.
A task is usually not directly completed when the DoD is met. The yearly developer survey conducted by StackOverflow has discovered, that over 95% of professional developers use some sort of version control system [Sta]. After producing new code and satisfying the project specific DoD, developer usually check their code into a version control system, such as GIT.

The presented research on task management in software development suggests, that there are numerous actions which are executed in every task. Project teams usually define the actions required before a task is completed in their project specific Definition of Done. These actions, e.g. version control interactions, are detectable within low-level IDE interaction logs and thus can be exploited for pattern discovery and predicting task switches.

2.4 FeedBaG Data

Proksch, Amann and Nadi have provided a general dataset for research on in-IDE activites [PAN18]. The data set contains low-level interaction data with added contextual information. This section will expain their data model and how this concept can be used for task boundary detection.

2.4.1 Data Structure for in-IDE Interaction Records

Interaction data collected from IDEs usually only describe what actions were executed by developers. They do not provide the event's context though. However, some actions require parameters or produce different outcomes based on the project's state. This data provides a lot of information about developer behaviour and is precious for interaction research.
Proksch has proposed *enriched event streams*. He suggests cultivating classic data streams with contextual information [Pro17]. The two-tier event scheme consists of a context and a process layer. The process layer denotes the invoked action, as well as its invocation time and duration. The context layer stores event specific, contextual data.
By having the additional contextual layer, events can be analysed much more precisely. In fact, detailed information about low-level events can totally alter their meaning. There is a fundamental difference between a successful project build and a failing one. Especially for detecting task boundaries, certain results of an action can influence the probability of a task switch in totally different way than other outcomes.
Proksch *et al.* have added a new concept to the context layer. Their so called Simplified Source Trees (SSTs) provide a condensed snapshot of source code. For certain events, SSTs are able to exactly depict what the code looked like at a certain point in time.

2.4.2 Events

FeedBaG collects a total of 19 different event types. Some of these events gather contextual data related to their source. However, not all of these events are suitable for our evaluation purposes. Therefore, this section evaluates which events can be used in order to gain knowledge about task

finishing patterns of developers.

```
public interface IIDEEvent
{
   string IDESessionUUID { get; set; }
   string KaVEVersion { get; set; }
   DateTimeOffset? TriggeredAt { get; set; }
   EventTrigger TriggeredBy { get; set; }
   DateTimeOffset? TerminatedAt { get; set; }
   TimeSpan? Duration { get; set; }
   IWindowName ActiveWindow { get; set; }
   IDocumentName ActiveDocument { get; set; }
}
```

Listing 2.1: C# implementation of IIDEEvent

Every generated event implements the IIDEEvent interface, which exposes the methods described in listing 2.1. Table 2.1 presents an overview of the events spawned by FeedBaG. A sophisticated explanation why certain events are meaningful for task boundary detection is provided below.

Table 2.1: FeedBaG events

Event name	Description
ActivityEvent	indicates activity within the IDE
BuildEvent	Builds started from Visual Studio
CommandEvent	triggered whenever an action is invoked, e.g. the press of a button
CompletionEvent	provides information about IntelliSense code completion
DebuggerEvent	provides information about debugger interaction
DocumentEvent	triggered when a file is saved, open or closed
EditEvent	provides information about edits of source code
FindEvent	indicates whether a find query was successful or not
IDEStateEvent	represents action which affect the IDE (e.g. startup, shutdown, window actions)
InfoEvent	used for logging and debugging purposes
SolutionEvent	triggered by solution creation, opening and more
SystemEvent	indicates system actions, e.g. sleep mode activation
TestRunEvent	provides information about testing related actions
VersionControlEvent	version control actions (commit, clone, ...)
WindowEvent	triggered by window focusing, close, open
NavigationEvent	provides information about source code navigation
ErrorEvent	stores information about errors thrown by FeedBaG
Deprecated Events	
InstallEvent	Installation of dependencies
UpdateEvent	Update of dependencies

Explanations

Some of these events contain valuable information for task boundary detection patterns. This section will provide insight on which information these events provide and why the events are relevant for our purposes.

TestRunEvent Based on the previously described ideal software development process, a task is not finished before all tests are successful. Therefore, test runs might be able to provide indications for both, positive and negative cases regarding task switches. The TestRunEvent states for every test whether it was successful, ignored or failing.

BuildEvent Most software products require that the source code is converted into a deployable artefact. This step is called building. When source code contains syntactical errors, builds fail. A task should never be finished with a defect build. Therefore, the BuildEvent provides valuable information.

VersionControlEvent Theoretically, the development part of a task finishes with the insertion of the generated code into the version control system. Therefore, a commit might be a good indicator for a nearing task switch. The VersionControlEvent captures interactions with GIT. It registers numerous actions besides commits, e.g. branching and cloning.

CommandEvent CommandEvents are spawned whenever a command execution is requested, e.g. by pressing a button. CommandEvents cover a wide variety of commands which can be relevant. E.g. certain build events are unfortunately only caught by the CommandEvent, but not the BuildEvent itself. Based on the CommandId, relevant events can be identified.

DocumentEvent It is reasonable to assume that some document manipulation happens at the end of a task. At the bare minimum, the documents have to be saved. It might also be recurring pattern that developers close task-related files after the task is completed.

While the remainder of the events might not explicitly provide information related to task switching, they certainly hold some informational value for the evaluation. Therefore, these events are not ignored but most likely aggregated in values such as number of events in a period of time or similar.

2.5 Machine Learning

Section 2.2 provided insight on how present research detects task boundaries. The majority of these approaches apply machine learning in order to detect task switches. We have also discussed an approach proposed by Shen which classifies task boundary detection as a supervised machine learning problem [SLDH06]. This chapter introduces concepts and applications of machine learning and data mining, and thus provides a foundation for the following explanations of the approach.

Research on self learning programs goes back into the last century. Samuel was able to show that computers are able to outperform humans in a short period of learning time. He programmed a computer to autonomously learn how to play the game of checkers [Sam59]. His efforts were some of the first modern approaches to self-learning programs.
Machine learning consists of a collection of countless algorithms with numerous different approaches and goals. All these techniques automatically create a model that can describe the relationship between given sets of input and output variables. Such models are built by *training* an

algorithm on a data set [BÖ14]. Subsequently, the model is validated against a data set which is not part of the training set.
Machine learning consists of mainly two categories: *Supervised learning* requires labelled data, i.e. contain independent and dependent attributes. The dependent attributes are called label. *Unsupervised learning* on the other hand does not require a label. Algorithms which apply unsupervised learning detect patterns in data autonomously [Gha04]. We will treat task boundary detection exclusively as a supervised learning problem. Hence, we will introduce supervised algorithms in the succeeding section.

Supervised Learning

The ultimate goal of task boundary detection is building a model which is applicable for pattern discover in low-level event logs. Therefore, we have a concrete goal we want to achieve. For such goal-oriented machine learning problems, supervised learning is frequently applied. Supervised algorithms build a function that describes the relationship between a set of independent values and a dependent value [MRT12]. The set of independent values is called *feature vector*, the dependent value is called *label*. Based on a labelled training set, supervised algorithms build a model which predicts the label corresponding to a feature vectors [MRT12].
There are mainly two types of values that can be predicted. Consequently, there are also two types of supervised machine learning algorithms. *classification* methods predict discrete values, *regression* algorithms calculate continuous ones.
Detecting task switches is a binary problem. A feature vector can either indicate a task switch or not. Therefore, task boundary detection is a classic classification problem. We made use of one of those classification algorithms. Logistic regression has been applied to many problems and is proven in the field of machine learning as effective when applied correctly. The following paragraph will explain the algorithm further.

Logistic Regression Logistic regression deals with modelling the relationship between a categorical variable and one or more input variables. The input variables can be either categorical or continuous. Based on a feature vector, logistic regression calculates the probability for the label to be of a possible outcome class. Logistic regression can therefore show, which outcome is the most likely for the set of provided input values [PLI02]. Due to the result being a probability value for an input class, logistic regression in fact is a method which calculates a continuous value, thus satisfying the definition of a regression algorithm. However, implementations of logistic regression usually return the most probable class value for the label, therefore being a classification algorithm.

2.5.1 Weka

There are many machine learning tools that support researchers in data analysis. These tools often times require only basic knowledge of machine learning and thus let users focus on their core research. This section will introduce WEKA, one of the most popular machine learning programs.
The University of Waikato has developed a workbench for machine learning, called WEKA [HDW94]. Since then, WEKA has gained enormous popularity. Over the years, WEKA has been expanded and optimized [HFH+09]. It comes with an easy-to-use user interface and is also available as a library for Java [Mav]. WEKA is extremely suitable for bulk analysis since the model building

and evaluation can be automated with minimal programming effort.
Additionally, WEKA provides hooks for all stages of data analysis. Data sets can be preprocessed with filters. The machine learning algorithm can be easily altered. Besides logistic regression, WEKA provides implementations of countless additional machine learning algorithms.
The flexibility and wide variety of machine learning algorithm implementations make WEKA powerful and extremely useful for our research. Chapter 4 will describe how WEKA is used in order to predict task switches and to evaluate the effectiveness of the built model.

2.6 Discovering Patterns in Application Logs

When software applications run in production, developers can not easily reproduce errors and bugs. To help with the reproduction step, applications are usually enriched with logging functionality. Logs provide insight on which actions were executed by users leading up to an undesired behaviour. Such logs however, can not only be used for reproducing bugs. They can also be analysed to understand how customers use the application. Especially web logs have been targeted by researchers with the idea to provide a simple technique to understand how applications are used. This insight allows developers to increase the usability of their applications.
Logs provide an immense amount of data. In order to analyse such masses, data mining techniques are frequently applied. Cooley *et al.* have described a specialised data mining technique which is tailored for analysing web logs [CMS97]. The so called web mining combines data mining with a variety of other techniques, e.g. machine learning [RSR09, SCDT00].
Web mining is divided into three categories: Web content mining, web usage mining and web structure mining [KB00]. Especially web usage mining has caught the interest of many researchers. The process of web usage mining is divided into three steps:

1. Preprocessing of data
2. Pattern Discovery
3. Pattern Analysis

The application of these steps is not limited to the analysis of web logs. In fact, they can provide insight on any interaction record when applied correctly. The following paragraphs will discuss what these steps consist of and how they are applicable to other problem domains.

2.6.1 Data Preprocessing

The preprocessing step is very important in order to prepare raw logs for web usage mining. Preprocessing consists of four tasks: Data cleaning, integration, transformation, and reduction [Datb]. These steps allow raw data streams to be evaluable by machine learning algorithms such as logistic regression. This section will explain these steps in detail.

Data Cleaning Data sets often contain impurities. Entries can be missing, irrelevant or inconsistent. Data cleaning deals with multiple data quality problems. Data points can violate integrity constraints, e.g. a negative value when only positive integers are allowed. Rahm and Do have suggested following workflow to deal with such problems [RD00]:

1. Data analysis
2. Definition of transformation workflow and mapping rules

3. Verification of the workflow
4. Transformation
5. Backflow of cleaned data

These steps allow data sets with quality problems to be improved and made ready for further (pre)processing steps.

Data Integration When multiple data sources are used, the schemes seldom are completely equal. However, in order to apply machine learning algorithms, the data has to be presented in a uniform manner. Data integration relies on value mapping and manual addition of missing values whenever possible in order to build a consistent data set [Data].

Data Transformation Data can be transformed in mainly three ways: Normalization, aggregation, and generalization.
Normalization is the process of converting values into a normalized form. This modification must be consistent, i.e. a specific value is always transformed to the same value. Sola and Sevilla have shown, that adequate data normalization can reduce estimation errors and improve overall performance significantly [SS97].
Data aggregation summarizes multiple values into a single one [Whaa]. Therefore, feature vectors decrease in size and meaningless variables are filtered out or combined into more meaningful ones.
Data generalization provides a summarized view on data. Basically, the low level data points are transferred into a higher, more abstract representation. This provides a quick overview of data and its context [Whab].

Data Reduction Data can contain a lot of non-essential values. Data reduction deals with removing this overhead. There are three ways this can be achieved: Reducing the number of attributes, reducing the number of attribute values and reducing the number of tuples. The latter is also called sampling.
Sampling is an important technique in order to deal with imbalanced data sets. Imbalanced data emerges when one class of an attribute appears often and another one very rarely [HG09]. This is especially disturbant when the imbalanced class represents the label. There has been extensive research on rare event classification. When machine learning is applied to highly imbalanced set, the precision and recall for the majority class are very high. However, the respective values for the minority class are very low [HG09].
Buda has conducted intensive research on the imbalanced class problem and came to the conclusion, that oversampling the minority class is the best method to gain meaningful results out of strongly imbalanced data sets. He goes even further and states that oversampling should erase the imbalance completely [BMM17]. There are various approaches to this principle. While some techniques rely on weighing minority class data points more, other methods generate additional data points for the minority class. These data points can either be copies of existing ones or newly projected vectors.
On the other end of the spectrum, undersampling erases imbalance by removing elements of the dominant class. A simple but effective implementation of this concept is randomized undersampling. As the name suggests, the representatives for the majority class are chosen randomly [KMM+07].

Chapter 2.3 provided a definition for tasks in agile development. Task duration should roughly correspond to the interval in which agile meetings are held. This means, that task switches happen rarely. In order to detect task boundaries, the data must be labelled with task switches. Therefore, we have to deal with a highly imbalanced data set. The techniques discussed in the previous section provide an approach to deal with this problem.

Feature transformation

The techniques introduced so far deal with quality problems in the data set. However, they are not sufficient for providing log entries with the needed information for a machine learning approach to pattern discovery. Chapter 2.5 introduced the concept of features. The label can be predicted based on these attributes. For this purpose, an additional technique must be introduced: feature transformation.
Choosing the right features for a problem domain is crucial in order to build successful models. When feature vectors only contain irrelevant attributes, the results provided by machine learning algorithms are equally useless. This concept is often referred to as *garbage in, garbage out* [CA18]. Classification algorithms map the relationship between a set of attributes and the label. Features that do not have any relationship to the label at all are thus unnecessary overhead. They might also distort the results. Feature transformation provides techniques to deal with these problems.
There are two types of feature transformation. *Feature construction* adds new features to a set of existing features. These additional values are constructed by performing logical operations on multiple, existing features within a vector. Basically, feature construction adds information about relationships between features [LM98].
Feature extraction on the other hand replaces existing features with new ones by applying a functional mapping to the values [LM98].
These two techniques can add valuable information to otherwise meaningless data. With feature construction, it is possible to add contextual data to raw logs while feature extraction transforms existing features into new, more promising attributes.

Preprocessing Summary

In this section, we provided an overview of the elements of preprocessing. Data cleaning, data integration, data transformation, and data reduction alter raw log data in such a way, that it can be evaluated with machine learning algorithms. Additionally, feature transformation changes and extracts features from provided data. After preprocessing, the data set is in a state which allows pattern discovery.

2.6.2 Pattern Discovery

Chapter 2.3 showed, that a task's progress can be described by the status of a clearly specified Definition of Done. This includes some recurring actions, e.g. test runs. It is therefore reasonable to assume that there are discoverable patterns in IDE interaction logs which lead up to task switches.
Application logs often only provide an incomplete and cluttered view on user behaviour. Web applications are able to serve multiple user requests in parallel. This behaviour is also reflected in logs. Entries are not bundled by request and are thus intertwined with entries from other requests.
Gill has provided a scheme how machine learning can be applied in log analysis to detect patterns [Gil]. In his approach, data is collected from various sources from which relevant informa-

tion is extracted using a Bayesan algorithm. Machine learning subsequently classifies the resulting logs from the previous steps and creates a model which puts newly generated log entries into corresponding categories.
Section 2.5 already introduced classification. Eventually, classifiers try to detect patterns in a set feature vectors. Therefore, they suit this step perfectly.
Summarized, the pattern is built by applying a machine learning algorithm to the preprocessed data sets. Logistic regression returns a set of weights according to which the probability for each class value is calculated. With the extracted model, it is subsequently possible to predict the label for feature vectors.

Summary

This chapter provided an overview of the current state of task detection within IDE interaction logs. While there are many tools that track the actions developers execute in their IDE, there is little research on putting the interaction in its higher context.
We have shown that task boundary detection has already been applied successfully to computer desktop work. Most approaches focus on the active windows and the artefacts users modify. Since IDEs are an all-in-one product, the former approach can not be applied to automated task detection in software development. However, the actions executed within the IDE can provide valuable metrics which indicate a task switch.
Most of the research conducted on task switch detection has made use of machine learning. Therefore, we provided a quick overview of machine learning techniques. We established that classification algorithms can be applied to preprocessed log data to detect patterns.
Additionally, FeedBaG and its data structure was introduced. By enriching low-level events generated from IDE interaction, FeedBaG provides contextual information which can be used for pattern discovery.
Finally, we have introduced web usage mining. By following a three step process - preprocessing, pattern discovery, pattern analysis - usage patterns can be discovered in web logs. We intend to adapt this process in order to detect task switches in IDE interaction logs.

Chapter 3

Automated Task Boundary Detection

Chapter 2 discussed how logs can be used to get insight on user behaviour. Task boundary detection in IDE interaction logs after all is a log analysis problem. This section provides an explanation on how task boundaries can be detected from interaction logs.
Section 2.4 introduced the data set we are using in our approach. FeedBaG's data is an enriched event stream which consists of standard log events with added contextual information. Throughout the description of our approach, we assume working with an enriched event stream. Section 3.1 explains what additional characteristics a data set must provide in order to be usable for task detection purposes with our approach.
Web mining is a technique explained in section 2.6 that is used in order to detect patterns in application logs. It is possible to adapt this algorithm with some modifications to analyse IDE interaction logs. The skeleton of the algorithm stays the same.
Section 3.2 introduces the preprocessing used in our approach. The raw data stream is transformed into a data set which is suitable for logistic regression. We cover three major steps: Feature construction, sampling and data cleaning.
Section 3.3 explains how models can be built from the set of feature vectors produced by the preprocessing step. Additionally, we shortly introduce the idea of pattern analysis, a way to draw conclusions from the extracted models.
In section 3.4, drawbacks and limitations of our approach are discussed.
Figure 3.1 summarizes the steps that are needed in order to convert a data stream of low-level actions into a model.

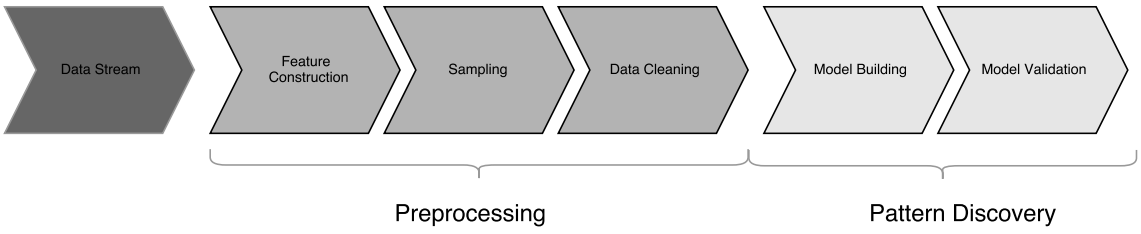

Figure 3.1: Steps for Automated Task Boundary Detection

3.1 Data Set Requirements

While in theory any type of interaction log, generated by an IDE, is suitable for analysis, there are certain aspects which simplify the process of data mining. This section provides an overview of characteristics a data set ideally features in order to be suitable for detection of task boundaries. However, analysis can also be conducted on unfitting data sets. For such cases, preprocessing must fill the gaps.

Ideally, the log exclusively consists of programming related events. Next to interactions within IDEs, developers use numerous auxiliary tools which are directly associated with software development. Version control, for instance, has become an integral part of software development. To draw a complete picture, interactions with such tools should be logged as well.
The dataset moreover should be ordered chronologically. Logs are usually sorted by the time an event is triggered at. It makes sense to emphasize this characteristic nonetheless. After all, our approach strives to predict task switches based on features constructed from raw event logs. Therefore, the actions leading up to a task switch must be known. This is only achievable with a chronologically ordered set.
Chapters 2.4.1 and 3.2.1 explain the importance of context information for machine learning. For our approach the data set must comply with this enriched data scheme. Without contextual information, the data set can only provide an incomplete snapshot of the state. As established in chapter 2.4.1, contextual information can completely change the meaning of a low-level interaction event. Our approach exploits this additional information to draw an accurate picture of the task's state.
Furthermore, the data should be bundled by developer. Minelly *et al.* have shown that while there are many similarities in IDE usage by developers, there are also many differences [MML15]. It is consequently reasonable to assume that software development is a developer-specific process. Therefore, it is crucial that each developer's behaviour can be studied separately. Having individual data sets also provides additional hooks for analysis. With developer specific information, it is possible to examine how developer behaviour differs and whether there are mutualities which allow a global task boundary detection model.
The data set should also provide information about programming sessions. This allows to evaluate whether developer behaviour stays consistent over multiple sessions. Ying and Robillard have shown that developers are influenced in their behaviour by the task's type [YR11]. Consequently, it is possible that there are other influences which affect how developers complete their tasks. Disruptions and additional knowledge could inherently change how the developer approaches a task. Such interactions usually happen in between in-IDE sessions. Having a way to distinct sessions in logs consequently is convenient for comparisons at a developer-individual level.

Label The most important characteristic the data set must provide is the presence of a task related label. In order to create a ground truth, there must be an event present which provides insight on tasks. This might be problematic. Task management is an abstract concept and is usually not done within the IDE. Most data streams do not track interactions with task management tools such as Trello or Jira. In such cases, the data stream has to be enriched with task meta data. It is sufficient to know the points in time at which a task switch was conducted.

Summary

This section presented and motivated the requirements for an event stream. These properties are needed for the data set to be evaluable using our approach. In summary, the data set should have these characteristics:

- Data concerns only in-IDE interactions and development related actions
- The data set is consistent
- The data set is ordered chronologically
- The evets provide contextual information
- The events are collected per developer
- The data set provides a task related entry

While contextual data can not be added in hindsight, the remaining characteristics can be achieved by sophisticated preprocessing. Section 2.6.1 already introduced methods used in preprocessing. Section 3.2 provides a more in-depth overview of the preprocessing techniques applied in our approach.

3.2 Data Preprocessing

As explained in the introduction of this chapter, the data stream has to be preprocessed before it is suitable for pattern discovery. Data preprocessing has multiple responsibilities, all of which are be explained in this section.
Our approach to data preprocessing consists of three steps. First, the low-level action data stream is converted to a higher-level data stream that describes the task's state. Section 3.2.1 explains how this is achieved.
In a subsequent step, the data is sampled. We are using a time based method which is motivated and explained in section 3.2.2.
Finally, the data has to be cleaned such that pattern discovery algorithms can be applied to it. Section 3.2.3 explains how data quality problems are dealt with.

3.2.1 Feature Construction

As chapter 2 explained, raw event logs can not draw a complete picture depicting the task's state. They accurately describe what happens but do not provide a description of a higher abstraction. Therefore, we propose a layer above the event stream which can be used in machine learning later on. Based on the low-level events, the newly constructed *state layer* generalizes the event stream to a stream of states.

State Our approach focuses on describing the task's state. A state is a simple collection of features which presumably indicate the progress made during the completion of a task. The state is computed by sequentially analysing each event contained in the event stream. Features are constructed by listening to certain events. For example, a feature that describes whether the last build was successful analyses build events and updates its value for each of them. On the other hand, there might be a feature that simply computes the active time. Such features consequently have to take every event into consideration. Consequently, an internal state describing the task's progress must be kept. Figure 3.2 visualizes how low-level events are converted into a higher abstraction.

By using such an abstraction approach, it is possible to predict task changes based on task states rather than low-level events. This step produces a lifeline whose state can be examined at

any active point in time.

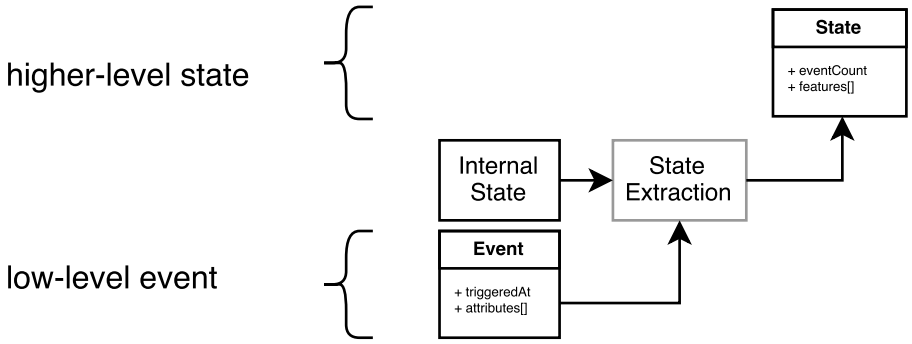

Figure 3.2: Extracting the Task State from Events

The state can consist of an arbitrary number of features. It is important, that all values are calculated for every active point in time. Activity is indicated by events. Consequently, the state is computed whenever an event is fired. Since events are fired very frequently, the state can be sampled in a periodic manner. Section 3.2.2 discusses this concept more precisely.
The next paragraph explains which attributes we decided to include in our state description. However, these are only examples. The feature vectors ultimately can contain any extractable attribute.

Features As explained in chapter 2.3, the Definition of Done usually relies on a project specific declaration. Nonetheless, there are certain characteristics that are likely to be present before every task switch. In our experiments, we will try to find relationships between task switches and the answers to the following questions:

1. Was the last build successful?
2. Was the last test successful?
3. How much time has passed since the last task switch?
4. How active was the developer within a certain amount of time?
5. How many files have been closed and saved in a certain amount of time?

Features 1 and 2 are motivated by usual quality control. Committed code should never be defect. Unit tests are a practical approach to guarantee correctness of program parts in a laboratory environment. Building the module on the other hand ensures that there are no compile errors. Ideally, developers conduct quick integration tests by running the application. Both, the last build and the last test, thus must succeed before a task switch.
Realistically, the probability of a task switch increases steadily. The more time has passed since the last task switch, the more likely is a nearing one. Feature 3 represents this idea.
Whether developers are rather active before task switches or not can be another indicator. Both of these cases are possible. Ying and Robillard have classified programming tasks into three classes: edit-first, edit-last and edit-throughout [YR11]. Edit-last tasks indicate a lot of activity in the

period leading up to a task switch. Edit-first tasks on the other hand would show little in-IDE activity before a task switch. Activity can be measured in two ways: Either, all actions executed within a defined interval are counted or the ratio of active time during an interval is calculated. After a task is completed, there is a need for clean up work. Within the IDE, a lot of files are opened during the coding process. Tasks are more or less independent, therefore successive tasks are unlikely to involve exactly the same files. Kevic and Fritz have shown that task boundaries can be reliably identified by comparing edited methods.Task switches mostly happen between intervals which do not share many related methods [KF17]. From that, it is possible to deduct that the majority of files used in a task are not needed in the succeeding one. Therefore, task switches might be preceded by an unusual amount of file closings. On the other hand, file savings presumably do not happen immediately before task switches due to the need for testing and building after source code modifications. This aspect is represented by the last feature.

3.2.2 Sampling

As a result of the rarity of task switches during software development, the data set is highly imbalanced. Therefore, we must deal with a problem known as overshadowing which is explained below.

Overshadowing Our approach suffers heavily from imbalance. On one hand, the probability that an interval contains a task switch is generally very low. On the other hand, certain state attributes change only infrequently. A flag which indicates whether the last test was successful switches at most whenever a test is run. However, if the sampling interval is well-chosen, successful tests become a strong indicator for task switches. Figure 3.3 visualizes this problem. With a short interval, the time between the test and the task switch is sampled multiple times, thus outweighing the positive case. In order to prevent this, the amount of negative samples must be reduced by choosing an adequate sampling interval.

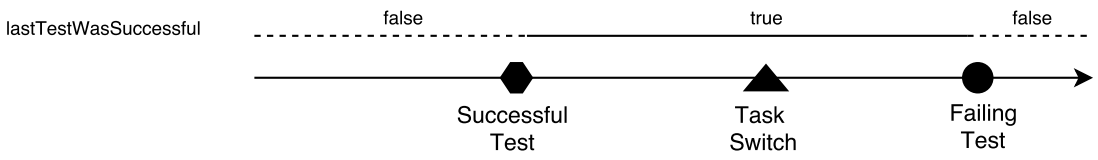

Figure 3.3: Overshadowing visualized

Chapter 2.6.1 hinted at sampling techniques. Undersampling is a very convenient yet practical approach for dealing with this outweighing problem. Due to the chronological order of events, the data stream can be considered to be a time line. Consequently, it is possible to periodically extract the state and use it as feature vector. Figure 3.4 visualises this process.

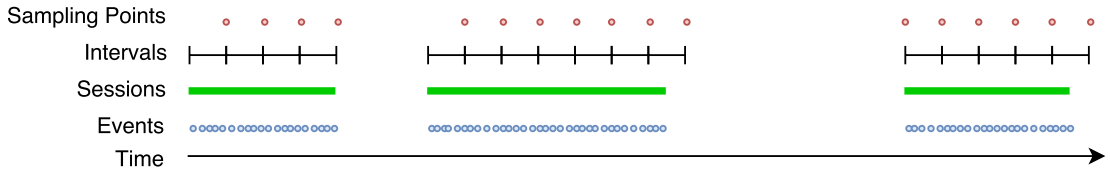

Figure 3.4: Interval-based Sampling

In general, undersampling techniques sample all instances of the minority class. The probability that a positive case appears exactly at the sampling instant in time is very low though. Therefore, it has to be ensured, that these cases are not ejected from the data set. Practically this can be achieved with two techniques: Positively labelled data points are always sampled. However, this leads to a ground truth including negative cases which are within a short time of a positive case and thus having similar values. This would water down the significance of the positive cases however. The second approach simply puts the positive case event in place of the succeeding sampled instance. This however may lead to overshadowing of positive cases when they appear closely after a sampled instance.

3.2.3 Data Cleaning

For our experiments, the data cleaning step must ensure two main characteristics of the data set: First and foremost, instances with invalid values have to be removed. Additionally, missing values have to be filled in.
Section 3.1 established, that our approach relies on a chronologically order data stream. Attributes that describe the time elapsed since a certain event require only positive values. For floating point ratio values, all instances must be between 0 and 1. Nominal values on the other hand must only accept values which are declared in the class definition. Unfortunately, it is not possible to correct invalid values. Therefore, data cleaning must remove all data points which violate at least one of these rules.
Missing values are an additional problem. Similarly to invalid values, data gaps can only be dealt with by removing affected instances. Values can only be inferred from other values when they are dependent on other attributes. However, for such cases it must be evaluated whether this dependent attribute can be removed from the feature vector.

Invalid or missing values are identified by iterating over the feature vectors once all data points have been created.

Preprocessing Summary

This section provided an approach to data preprocessing. In three steps, the low level event stream is processed into a higher level abstraction of the tasks state at any active point in time. By keeping an internal state and evaluating each event chronologically, the state can continuously be updated. In order to prevent overshadowing caused by data imbalance, the states are only sampled periodically. In the final step, data points with invalid values are removed.

3.3 Pattern Discovery with Machine Learning

Classification algorithms discover patterns in data sets. As explained in section 2.2, task boundary detection can be treated as a supervised machine learning problem. Consequently, any classification algorithm, that takes labelled feature vectors as input, can be used for task boundary detection.
The previous step, preprocessing, prepared the data in order to comply with the requirements posed to data which is used in machine learning. The low-level event log was transformed into a higher level depiction of the task's state. Therefore, the foundation for pattern discovery with a classification algorithm was built.
For our approach, logistic regression was chosen as preferred machine learning algorithm. Section 2.5 introduced this technique. Logistic regression is often used in similar problems and can decently deal with rare event data. While we will use exclusively logistic regression for further evaluation purposes, our approach is also applicable with other supervised classification techniques such as decision trees or support vector machines.
The actual machine learning process consists of two steps: First, the data set has to be divided into multiple parts. The majority of these subsets are used in a training run. Based on this training data, the machine learning algorithm subsequently produces a model which describes the relationship between the label and the features.
In the second step, the model is validated against the remainder of the subsets. Basically, the model classifies each instance contained in the validation set. Each classification is then compared to its actual label. The validation step collects all comparisons and evaluates how accurate the model is.

Pattern Analysis

The final step concerns the understanding of the collected patterns from the previous step. Not all patterns are applicable to every problem domain. For example, a live prediction of tasks must not rely on future events. However, analysing an event stream statically does not rely on this time constraint and therefore can also make use of characteristics appearing after a task was finished. Pattern analysis is therefore important in order to find patterns which fit its purpose [IV06].
Logistic regression assigns a weight to each feature. The weights indicate how much an attribute influences the probability of a certain label value. Due to this knowledge, the model can be analysed and tendencies can be identified. Supposedly, there are certain features which have a higher impact on the label's value than others. Pattern analysis provides a step for such examinations.

3.4 Limitations

The previously suggested approach comes with some limitations. This section provides a quick overview of the drawbacks of the proposed methodology.

Task Completion Process Pattern discovery is only applicable if there are recurring sequences of actions which lead to a task switch. This approach only works for developers that religiously follow the same work flow over and over again. Changes in behaviour can most likely not be compensated immediately. Predicting task switches solely based on the actions the developer executes, might not lead to a result. Ying and Robillard were able to show that the type of task influences the way developers behave when completing a task [YR11]. Their research focuses on

the timing of the coding period during a task. The proposed approach would most likely struggle with the *edit-first* task completion style[1].

Task Definition The definition of a task is very loose and not clearly declared. The scrum alliance defines a task as "a unit of work generally between four and sixteen hours" [Sza07]. In reality, developers report their task taking from 30 minutes up to multiple days [MBM+17]. Eventually, defining the contents of a task is up to each project team.
The discrepancy in task duration poses an additional threat to the results. For any time based feature, the length of a task might influence the accuracy of the machine learning algorithm negatively.

IDE Centricity The approach of evaluating IDE generated logs in order to predict task boundaries ignores the fact that a task is not only made up of coding. Perry *et al.* have shown that the majority of time used for a task is not spent with programming but is from organisational nature [PSV94]. Collecting data from within the IDE thus ignores certain aspects of a task. Some activities, like code reviews, might interfere with the perceived time spent on a task and thus makes it hard to reliably detect task switches. Time based features are consequently influenced by coding unrelated activities. These activities are not account for in our approach.

Incomplete Task Switching There are cases in which a developer is not able to completely perform a task. This might be due to missing knowledge or dependencies which can not be resolved straight away. For such cases, tasks are put aside and picked up later. Our approach can not deal with such interruptions as the recurring steps, e.g. testing, are not necessarily successfully conducted before the task is put on hold.

[1] When the majority of edit events are executed during the first half of the task, the style is denoted as *edit-first* [YR11].

Chapter 4

Implementation

We developed an implementation of the approach explained in chapter 3. This chapter explains the important parts our stream analysing tool. Based on this tool, we conducted several experiments, explained in chapter 5, that evaluate the effectiveness of our approach. Our implementation prepares the low-level event stream into a format which can be used for model building with WEKA.

4.1 Architecture

The implementation consists of three modules: Section 4.1.1 shows how the *EventParser* deserialises FeedBaG's data stream. From there on, the objects are written into the *EventQueue*. Section 4.1.2 describes and motivates the functionality of the *EventQueue*. Finally, section 4.1.3 explains how the *EventQueue*'s contents are preprocessed by the *QueueProcessor*.
The resulting three tier architecture allowed parallel deserialisation and stream processing of the events. Figure 4.1 provides an overview of the architecture.

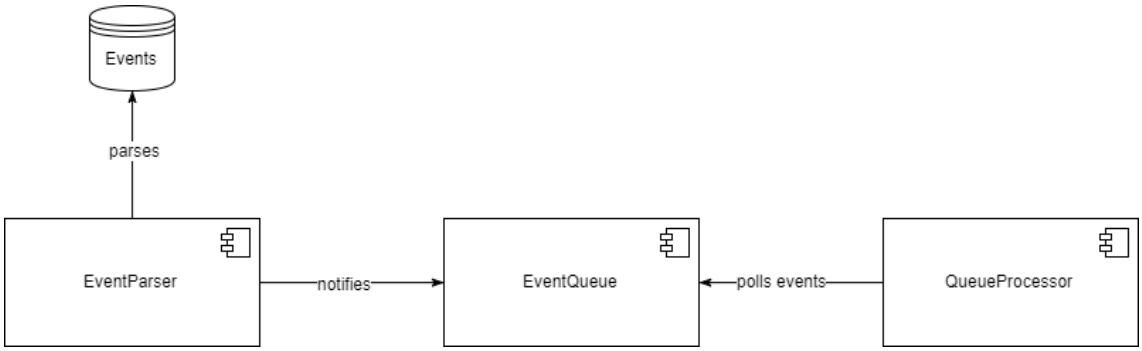

Figure 4.1: Architecture

4.1.1 EventParser

The data provided by FeedBaG is stored in archives containing the events of a single developer. The events are serialised in JSON format, whereas each file represents one event. The files are ordered chronologically by the event's date.
The *EventParser* detects all zip files in a given folder and parses the events contained in the archives. Additionally, *EventParser* implements the *Publisher* interface which allows interested objects to subscribe to parsed events. The subscribers must implement the EventParsedListener interface. Whenever an event is parsed, all registered *EventParsedListener*s are notified with the *IIDEEvent*.
This approach is an implementation of the well-known observer design pattern. Figure 4.2 models this structure.

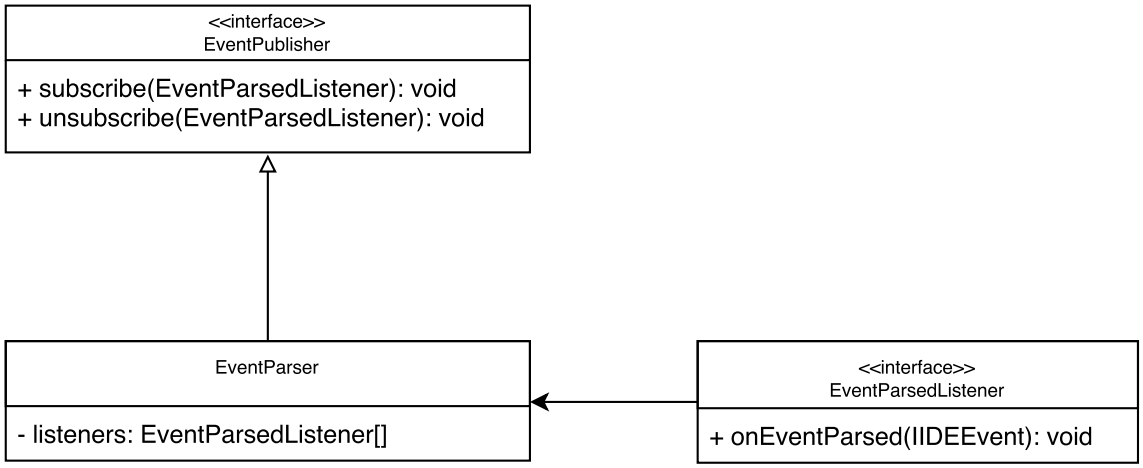

Figure 4.2: Class Diagram of EventParser and Listeners

4.1.2 EventQueue

The *EventQueue* is a simple first-in-first-out (FIFO) queue[1]. The *EventQueue* implements the aforementioned *EventParsedListener* and is subscribed to the *EventParser*. Therefore, whenever an event is parsed, it is stored in the *EventQueue*.
By using a FIFO queue data structure, the order of the events is preserved.
The *QueueProcessor* heavily relies on this ordering of events to construct features correctly. The EventQueue provides a simplistic interface which allows adding to and polling events from the queue (see listing 4.1).

```
public interface EventQueue {
   void add(IIDEEvent event);
   IIDEEvent poll();
   int size();
}
```

Listing 4.1: EventQueue interface

[1]The elements that are added first to a FIFO queue are also released first.

4.1.3 QueueProcessor

The *QueueProcessor* implements the preprocessing steps described in chapter 3.2. The processor exposes an interface which allows *Aggregator*s to subscribe themselves. *Aggregator*s construct features which are subsequently written into an event's feature vector.
The *QueueProcessor* periodically polls the *EventQueue*. Whenever an event is ready for preprocessing, all aggregators are queried. They are provided with the event and return a key-value pair which represents the attribute name and its value.
This implementation makes the *QueueProcessor* highly extensible. Features can be added and removed during run-time.
Additionally, the *QueueProcessor* has to be instantiated with a *SamplePicker*. Section 3.2.2 has explained that the event stream is highly imbalanced in terms of task switches. Therefore, an adequate sampling technique has to be implemented. The *SamplePicker* indicates for each event whether it should be sampled or not.
One could argue that for efficiency reasons, the *SamplePicker* should be queried first, such that the *Aggregator*s are only queried if the event should be sampled. However, the *Aggregator*s can keep an internal state and thus rely on the processing of every event in a chronological manner. For example, the *EventCountAggregator* counts every event that has been triggered since the starting point. Therefore, every event has to be provided to it. Otherwise, the values are incorrect.
The *QueueProcessor* collects all sampled feature vectors. At this point, the ordering is not important any more since supervised machine learning algorithms do not care about chronology. Subsequently, the feature vectors are converted into the WEKA specific ARFF file format and thus ready for the model building step. With that, the preprocessing of the event stream is concluded. Figure 4.3 visualizes the responsibilities of the *QueueProcessor*.

Architecture Summary

We provided an implementation of the approach described in chapter 3. Our approach heavily relies on an implementation of the publisher-subscriber pattern. The data stream is deserialised by the *EventParser*. The parsed events are written into an intermediate storage unit, the *EventQueue*. The *EventProcessor* queries the events from the queue and computes the state layer. Subsequently, the feature vectors are written into a WEKA compliant format, ARFF.
The parsing and preprocessing steps can run in parallel. Furthermore, objects are only stored in memory until they are processed. This guarantees a memory efficient stream processing.

4.2 Model Building and Evaluation with WEKA

The previous section showed how raw data logs are transformed into a WEKA compliant format. Based on these proceedings, it is possible to detect patterns in the data set using machine learning algorithms. Chapter 2 introduced WEKA and how it can be used in code. This section explains the algorithm that analyses batches of ARFF files using the same technique for each of them.

The previous step produces an ARFF file for each developer. The *Classification* class builds a model for every ARFF file in a given folder. WEKA allows the addition of multiple preprocessing steps as well as the declaration of the machine learning algorithm. The *Classification* class is highly modifiable. Preprocessing steps can be added at run-time, machine learning algorithms can be chosen and configured on the fly. By using this kind of composition, the application provides means for different evaluations and comparisons.

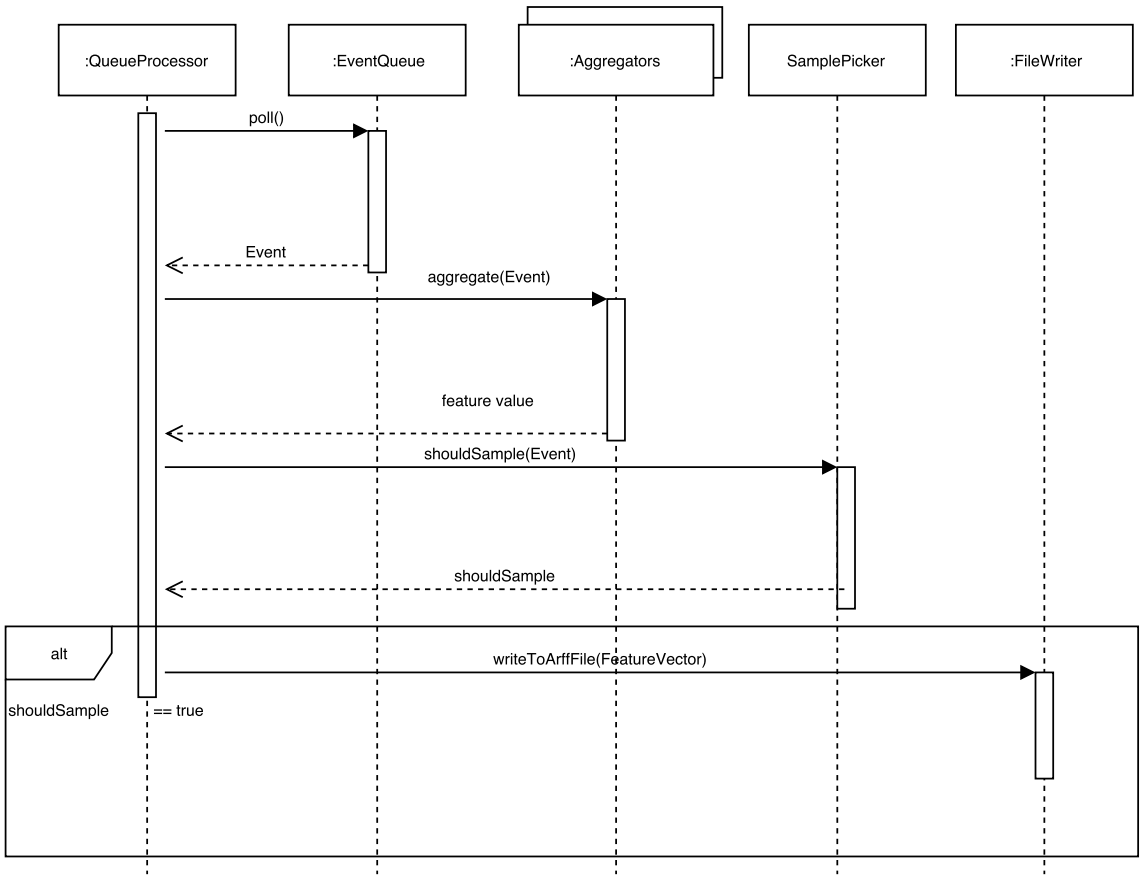

Figure 4.3: Sequence Diagram of QueueProcessor

WEKA provides numerous features beyond applying machine learning algorithms. In summary, the *Classification* tool takes care of these steps:

1. Data Cleaning - Remove invalid data
2. Applying additional sampling techniques
3. Model Building
4. Model Validation
5. Output of batch statistics

Machine Learning For our experiments, we used logistic regression. Chapter 2 has provided explanations and motivations about logistic regression. The algorithm requires a data set as input as well as some parameters. Logistic regression is capable of predicting the class of one attribute. Therefore, the labelling class has to be specified. Additionally, WEKA provides means for cross validation of a data set. For our approach, we applied a ten-folded cross validation of the data set. This means that the data set is divided into ten parts, nine of which are used as training set. The built model is validated against the remaining part. WEKA provides recall and precision values

for the label's possible values in a confusion matrix.

The gathered results are written into a simple CSV file. Therefore, the results are in a format which can be used by many visualization programs. It is fairly simple to gain a quick overview on the model's effectiveness by presenting the results in an adequate diagram, e.g. in a box plot.

Summary

This chapter provided an implementation of the approach explained in the previous chapter. The overall implementation consists of two components, preprocessing and evaluation.
EventParser, *EventQueue* and *QueueProcessor* transform the raw log data into feature vectors that describe the task's state at sampled points of time. The sampling is achieved by picking events in intervals. The results are collected and written into a file format which can be used by WEKA.
The ARFF files produced by the preprocessing step are evaluated by the *Classification* class. Filters and machine learning algorithms can easily be added and modified. Eventually, the results are written into a CSV file for further data visualization.

Chapter 5

Evaluation

The previous two chapters provided an overview of both, our approach and our implementation. In order to evaluate how precisely the presented methodology can predict task changes, we conducted several experiments. We focused on detecting task switches at developer level. Nonetheless, we also conducted experiments on a larger and a more granular scale. This chapter explains the three experiments we carried out. For this purpose, we aggregated the data in three different ways: First, we examined whether a global model for this data set can be built. Thereafter, we tried to build individual models. Finally, a session based model was analysed.

5.1 Data Set

FeedBaG has provided a data set containing interaction data from over 80 developers. In total, the event stream consists of more than 11M low level events [PNAM17]. This equals to around 15K hours of in-IDE interaction time. The majority of the contributing developers are professionals, while some of them are students or hobby programmers.
Section 2.4 introduced the events FeedBaG produces. The data violates one of the requirements posed to data sets in section 3.1. Unfortunately, FeedBaG does not track the user's task. Therefore, we decided to use commit events as task switches. Both, individual commits and tasks, should only consist of related work [Ver]. While commits presumably happen more often than task switches, their scope can be seen as subtask of a larger task. For these reasons, we considered commits to be an adequate task switch representation. Consequently, data sets, that do not contain any commits, must not be considered for evaluation. Due to this, the data set was reduced to the interaction data of 44 developers.
Surprisingly, only 16 of the remaining 44 developers have recorded at least one test. Testing plays a central role in our approach. The premise, that tasks must not be completed before all written tests succeed, can only be evaluated when tests have been run. We theorize that the absence of tests noticeably reduces the effectiveness of our approach. Therefore, the data set was reduced to 16 participants, still containing more than 4.5M events. For our second experiment, building developer individual models, we compared whether our assumption concerning the effectiveness of data sets with and without tests is correct.

Chapter 2 established an idealized work flow. The described process weighs unit tests and project builds heavily when analysing task boundaries. We assumed, that a task can not be finished as long as builds and tests do not succeed. Figure 5.1 visualises the frequency of the involved events, namely Commits, BuildEvents and TestEvents, across all considered data sets. It is identifiable, that there are more test runs than commits. However, a closer look on the data shows quickly,

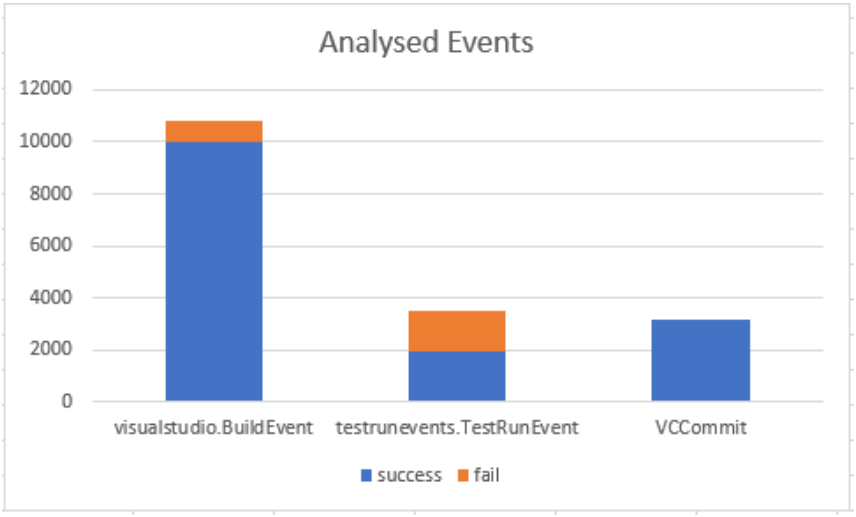

Figure 5.1: Comparison of events

that there are actually less succeeding tests than commits. Consequently, not all developers test successfully before they commit.

Intervals Throughout our experiments, the data was sampled using five different intervals:

1. 30 seconds
2. 1 minute
3. 2 minutes
4. 5 minutes
5. 10 minutes

This allowed us to examine the influence of the sampling interval on accuracy.

5.2 Feature Vector

Section 3.2.1 introduced feature construction. Additionally, it proposed numerous attributes which are able to explain the state of a task. Based on that, the following feature vector was built:

Table 5.1: Features used for boundary detection

Name	Type	Values
ActiveTimeRatioInInterval	continuous	0.0 - 1.0
EventsInInterval	discrete	positive integers
FilesClosedInInterval	discrete	positive integers
FilesSavedInInterval	discrete	positive integers
SecsSinceLastBuild	discrete	positive integers
SecsSinceLastCommit	discrete	positive integers
LastBuildWasSuccessful	categorical	't', 'f'
LastTestWasSuccessful	categorical	't', 'f'
BuildInInterval	categorical	't', 'f'
SuccessfulTestInInterval	categorical	't', 'f'

5.3 Global Model for Task Boundary Detection

A global model for task boundary detection would have many benefits. The model could be applied to live data without the need for an initial training. It could also serve as starting point based on which the model could be personalized for every developer individually.

Idea For this experiment, the machine learning algorithm is trained on a fraction of the developers and subsequently verified against interaction logs of the remaining developers.

Expectations It would be surprising if such a model exists. Software development probably is an individual process. While there are certainly a lot of similarities between developers, the discrepancies in developer behaviour make a global model very unlikely.

Approach The preprocessing step ensures that the data is consistent. Additionally, the data points are independent from each other. These characteristics permit to combine the data sets into a single one.
The data sets of the majority of the developers were merged into a single file. Subsequently, the classification tool built and validated a global model using logistic regression. The model was then tested against the remainder of the data sets. This followed the standard machine learning approach by dividing the data into a training and a test set.
For this experiment, the data of 13 randomly chosen developers was used as training data. The remaining three data sets were used as validation set. We tested the accuracy of these models using each of the five previously mentioned intervals. In total 15 evaluations were conducted, one for each interval and validation data set combination.

Results Table 5.2 summarizes the precision and recall values for both task switches (Pos) and non-task switches (Neg). During these tests, a total of 2 task switches were correctly detected, both of which were from the same developer. For the data sets of the remaining two test developers, not even a single task switch was correctly identified. Meanwhile, there were a lot of false negatives. This clearly indicates that not only it is not possible to obtain a global model. It is also highly unlikely that even one task switch can be predicted in any validation set. In fact, all sampling intervals below five minutes were unable to predict a single task switch correctly.
The results from the experiment with combined data sets clearly support our assumption that a

global model is not realistic. However, the results were even worse than expected and clearly deny any hope for a globally applicable model.

Table 5.2: Results for Global Model with 13:3 Split

Dev	precisionPos	recallPos	precisionNeg	recallNeg
		600s		
1	0.727	0.083	0.616	0.979
2	0	0	0.899	1
3	0	0	0.972	1
		300s		
1	0.5	0.008	0.672	0.996
2	0	0	0.93	1
3	0	0	0.983	1
		120s		
1	0	0	0.8	1
2	0	0	0.959	1
3	0	0	0.995	1
		60s		
1	0	0	0.864	1
2	0	0	0.974	1
3	0	0	0.996	1
		30s		
1	0	0	0.911	1
2	0	0	0.982	1
3	0	0	0.998	1

5.4 Individual Model per Developer

Having individual models for each developer provides the most value. Such models could be applied to live task boundary detection. Additionally, behaviour researchers often analyse each developer individually and aggregate the findings to draw an overview. For such studies, these models would provide valuable insights. However, personalized models are only feasible when a ground truth can be built. This requires the manual annotation of task switches.
Due to these reasons, we heavily emphasized on experiments regarding personalized models. First, we tried to build a model with the interval based sampling technique. Thereafter, we evaluated the influence of balancing out the data set. We were able to obtain each model and thus the feature weights. Based on this, we tried to find tendencies regarding which features influence the results more than others.
In an additional step, we compared the results gained from an example interval and the minimal data set with the data set of developers who do not test.

Idea For every developer, we built the state layer for each of the aforementioned intervals. We applied logistic regression with a ten-folded cross validation for each of them. Based on this, we were able to evaluate how accurately the models can predict task switches within the data set.

Expectations Detecting patterns in user specific behaviour appears likely. First of all, professional developers usually work for one employer. Software companies tend to enforce a clearly defined work flow. The defined rules can guarantee minimal quality features such as unit correctness. Additionally, projects usually declare a Definition of Done. This requires developers to execute certain steps for every task. Within these boundaries, developers can individually adjust their behaviour. Presumably, they settle for a personal work flow. These characteristics appear exploitable for task boundary detection.
Furthermore, it is expected that longer intervals provide better results. This is due to lower imbalance of the data set when taking fewer samples.

Results

We conducted multiple experiments with different sampling intervals. The results are summarized in table 5.3. All intervals showed an accuracy of at least 90%. This is not surprising, in fact if every state is predicted to indicate a negative case (i.e. not a task switch), the accuracy would be around 95%, depending on the frequency of task switches. In order to get a more meaningful picture, we evaluated F1 scores for both positive and negative cases, i.e. task switches and non-task-switches. The F1 score is the harmonic mean between precision and recall. Therefore, it can depict a combined view of these values. The F1 score can reach from 0.0 to 1.0, with 1.0 being the best possible value (indicating that both precision and recall are 1).
The F1 scores presented a large discrepancy between positive and negative cases. Especially the values for positive cases, i.e. task switches, were very low. Figure 5.2 visualises this difference. The median of F1 scores of positive cases was almost constantly at 0. This indicates, that for at least half of the developers, not a single task switch was detected correctly. This observation is clearly supported by the last column which states for how many data sets at least one task switch was recognized. For example, using a five minute interval, at least one task switch was correctly detected in 8 data sets. This is almost a textbook example for a simple problem logistic regression suffers from: data imbalance. In fact, the datasets showed ratios of positive to negative cases ranging from 1:5 up to 1:3500. Imbalanced data sets generally provide bad results for the minority class. This was clearly observable in our experiments.

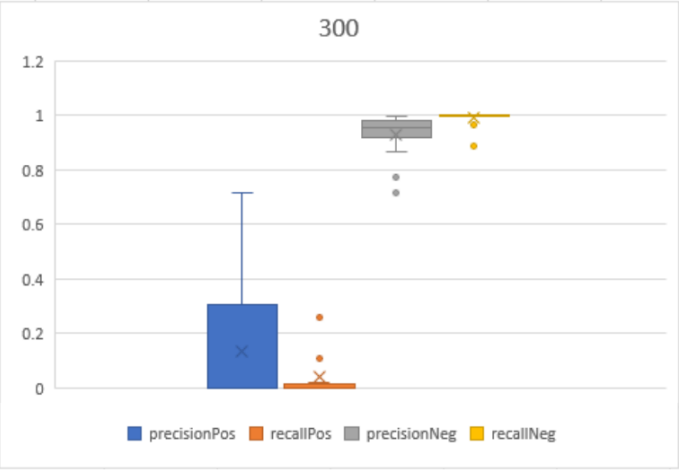

Figure 5.2: Mean F1 Scores without oversampling

Table 5.3: F1 scores for undersampled data set

interval	mean F1 pos	median F1 pos	mean F1 neg	median F1 neg	F1 pos > 0 [1]
30s	0.05	0.00	0.99	0.99	3
60s	0.08	0.00	0.98	0.99	4
120s	0.10	0.00	0.97	0.98	7
300s	0.12	0.01	0.96	0.97	8
600s	0.16	0.01	0.94	0.96	8

Dealing with the Imbalance We already conducted steps in order to reduce the set's imbalanced by using a periodic sampling. However, task switches only happen very rarely. The larger the sampling interval was chosen, the smaller was the imbalance. However, it it impossible to achieve a totally balanced out set with this technique. At this point, we experimented with various oversampling techniques. Luckily, WEKA provides natively some supervised filters. We tried two simple algorithms for this experiment. *ClassBalancer* removes imbalance by weighing the labels differently, such that both classes provide an equal total weight [Cla]. *Resample* on the other hand randomly subsamples the data set and replaces samples of the majority class with instances from the minority class [Res].
This adjustment improved the values of precision, recall and F1 value for task switches significantly in every data set. On the other hand, the respective value for negative cases decreased slightly. Figures 5.3 and 5.4 compare exemplary results using two oversampling techniques and a five minute undersampling interval. *Resample* was able to provide the most promising results. *ClassBalance* had slightly worse values and more extreme outliers. Both of these techniques provided precision and recall means for both classes above 70% however. Compared with figure 5.2, these results were better by orders of magnitudes. Surprisingly, the overall values were not correlating with the interval length. This is most likely due to the fact that all of the data sets are balanced out. Without oversampling on the other hand, the imbalance varies.

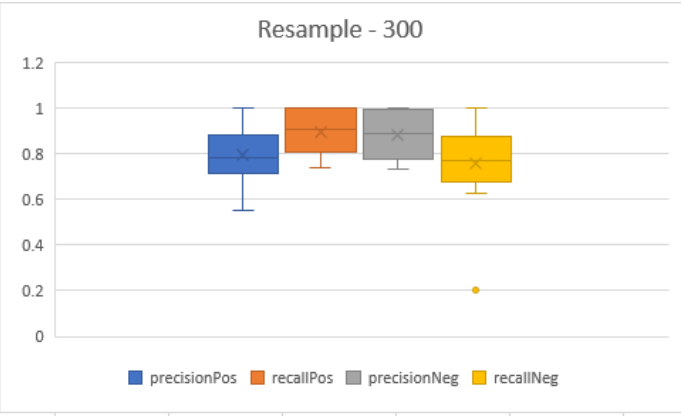

Figure 5.3: Mean F1 scores with Resample

[1] Indicates how many data sets contained at least one true positives

Table 5.4: F1 scores for data set with *Resample*

interval	mean F1 pos	median F1 pos	mean F1 neg	median F1 neg	F1 pos > 0
30s	0.90	0.87	0.87	0.88	16
60s	0.90	0.92	0.86	0.89	16
120s	0.91	0.90	0.85	0.86	16
300s	0.92	0.91	0.83	0.83	16
600s	0.86	0.86	0.82	0.85	16

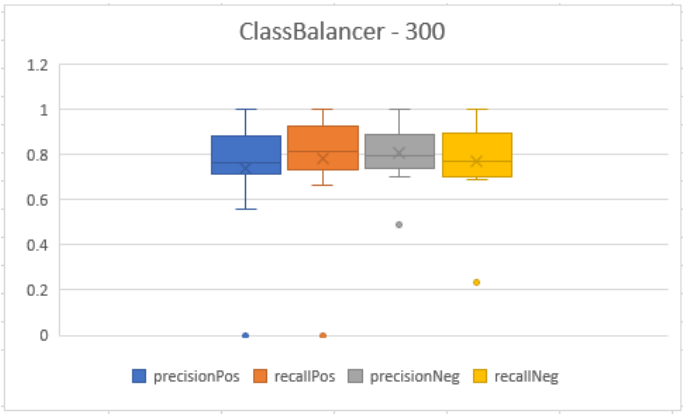

Figure 5.4: Mean F1 scores with ClassBalancer

Table 5.5: F1 scores for data set with *ClassBalance*

interval	mean F1 pos	median F1 pos	mean F1 neg	median F1 neg	F1 pos > 0
30s	0.83	0.81	0.83	0.85	16
60s	0.88	0.87	0.87	0.88	16
120s	0.88	0.88	0.86	0.88	16
300s	0.87	0.86	0.92	0.91	16
600s	0.86	0.86	0.92	0.93	16

Regression Threshold Logistic regression calculates the probability for a feature vector to obtain a certain label value. By default, WEKA returns the label's class value if its probability is higher than 0.5. This threshold can be altered and thus the results can be changed.
With oversampling, we introduced a new problem by predicting too many task switches, i.e. increasing the false positive rate. Increasing the threshold is a possible counter measure against the high false positive rate. We compared several thresholds and their influence on the accuracy of the model. Figure 5.5 visualizes the results gained from these tests.
Changing the threshold did not alter the results significantly. In fact, the ideal threshold is somewhere between 0.5 and 0.6. However, the accuracy did not change noticeably once the threshold was above 0.4. From this we can conclude, that the model is in fact not over-confident. Higher thresholds did not reduce the amount of false positives and thus increasing the needed confidence for task switch detection did not improve the results.

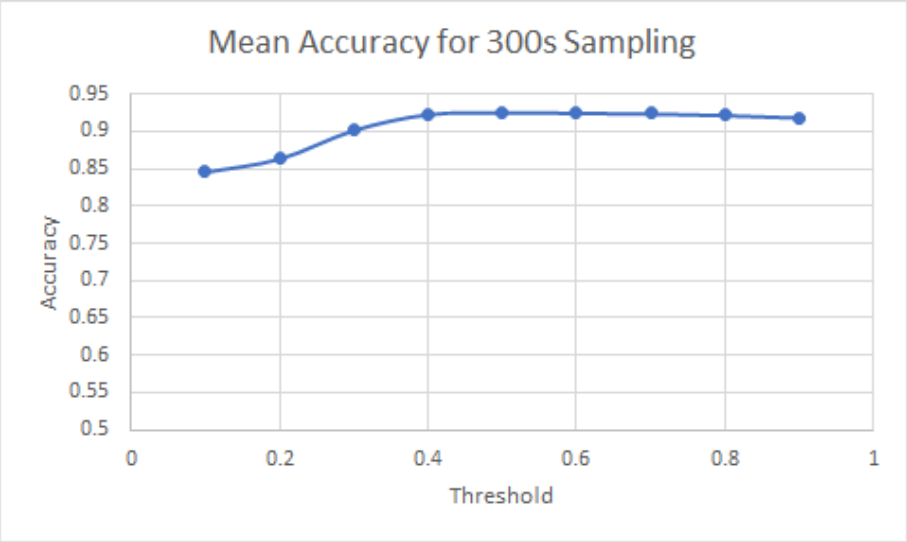

Figure 5.5: Threshold Influence on Accuracy

Feature Relevancy Section 5.2 explained which features we were using for our evaluation. Not all of these features had the same impact on the result. By analysing the distribution of the feature correlations, it was possible to examine which features were more heavily weighted and which attributes only influenced the results slightly. Figure 5.6 presents a sample model calculated by WEKA. The numerical values are the weights attributed to each feature.
In order to understand the significance of the weights we have to take a closer look on the logistic regression equation 5.1:

```
                                  Class
Variable                              t
=======================================
ActiveTimeInLast30s                0.35
EventsInLast30s                  0.9764
FilesClosedInLast30s             0.7869
FilesSavedInLast30s              2.0564
SecsSinceLastBuild                    1
lastBuildWasSuccessful=f         15.7575
Within30sOfBuild=f                0.143
Within30sOfSuccessfulTest=f      0.5157
lastTestWasSuccessful=f          1.0668
```

Figure 5.6: Sample model output

$$log(\frac{p}{1-p}) = \alpha + \sum_{i=1}^{n}(\beta_i * X_i) \tag{5.1}$$

β describes the weight of feature X, α is a constant. By solving the equation for p it is possible to obtain the probability for a feature vector to be labelled with a the given class value. If p is bigger than 0.5 it is likely for the label value to equal to the inspected class value.
The way WEKA presents this data is easily explained: A change of value X changes the log odds by the provided value. When *lastBuildWasSuccessful* changes it's value from f to t, the log odds of the value t for the label changes by 15.7575. This indicates that *lastBuildWasSuccessful* has a relatively high impact whereas a change for *Within30sOfBuild* has a minimal impact on the result.
We collected the data of all 16 developers in order to analyse whether there are certain values which have a tendency to influence the outcome strongly. Figure 5.7 shows an excerpt of the weight distribution for each value. Interval dependent features were aggregated into one class each.
There was a slight tendency that *lastTestWasSuccessful* had a positive influence on the log odds. In other words, when *lastTestWasSuccessful* was false, a task switch was less probable than when it was true. Similarily, *lastBuildWasSuccessful* also had a slightly positive influence on the outcome when it is true. On the other hand, *ActiveTime* had a negative influence. This means that task switches are more likely to be preceded by intervals with lower activity rather than high activity ones.
However, these values have to be treated carefully as there were heavy outliers. For example *FilesClosed*, which indicates how many files have been closed in the interval leading up to a task switch, basically only consisted of outliers. *WithinBuild*, which states whether a successful build was executed close to the task switch, had an outlier with the value of roughly 32.5K. Even *ActiveTime* had weights going below -20K.
Additionally, all values had both positive and negative weights. It follows that there was not a value, which must be present in order for a task switch to happen. We previously assumed that a task can not be completed without succeeding tests. However, this claim is not supported by the data gathered.

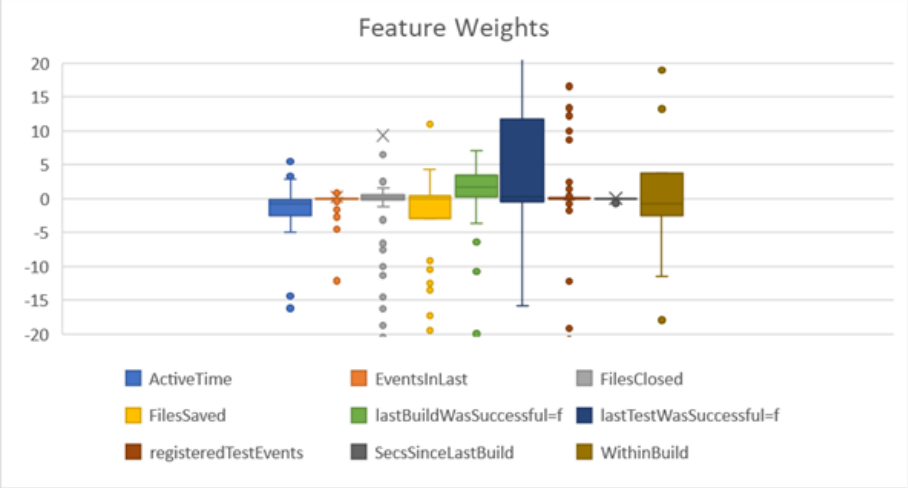

Figure 5.7: Distribution of feature weights

Influence of Tests Initially, we removed all data sets, that did not include any tests, since we assumed that testing is an integral part and a strong influence for task switches. We could partially confirm this idea by showing, that successful tests tend to have positive influence on the probability of an upcoming task switch. However, this had to be confirmed first.
For this purpose, interaction records which contained no tests were evaluated using the initial methodology. I.e., the states were aggregated for the same intervals using no oversampling technique. Figure 5.8 compares the F1 score for detecting task switches from data sets with test and without tests. Additionally, figure 5.9 shows the differences for the mean F1 score for negative cases and mean accuracy between data sets with testing and those without.
As we assumed, the results gained from developers who test frequently are better than those from developers who do not test. However, the differences are marginal. While having tests improved the F1 score for task switches by 4%-9%, the differences for the mean F1 scores for negative cases and for the accuracy are within 2% and 5% respectively.
It is not trivial to interpret these findings. While the previous section has shown, that successful tests are indeed a good indicator for task switches, the results for data sets without tests were not massively worse. However, this might be explicable by the fact, that apparently many developers test their tools by running them, not by using unit tests. Therefore, successful builds contribute heavily to the models.

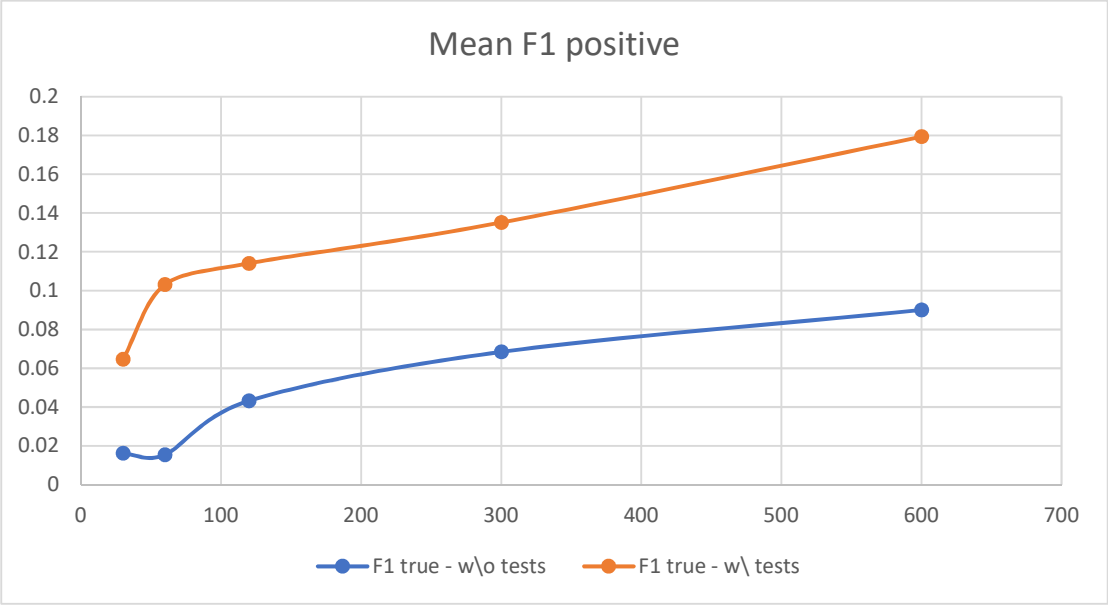

Figure 5.8: Mean F1 scores for positive cases without oversampling

Developer Model Summary

We conducted many tests regarding building a developer specific model. We obtained poor results for our basic approach with interval based undersampling. The outcome suggests, that overshadowing is an extreme problem when detecting task switches. This claim was supported

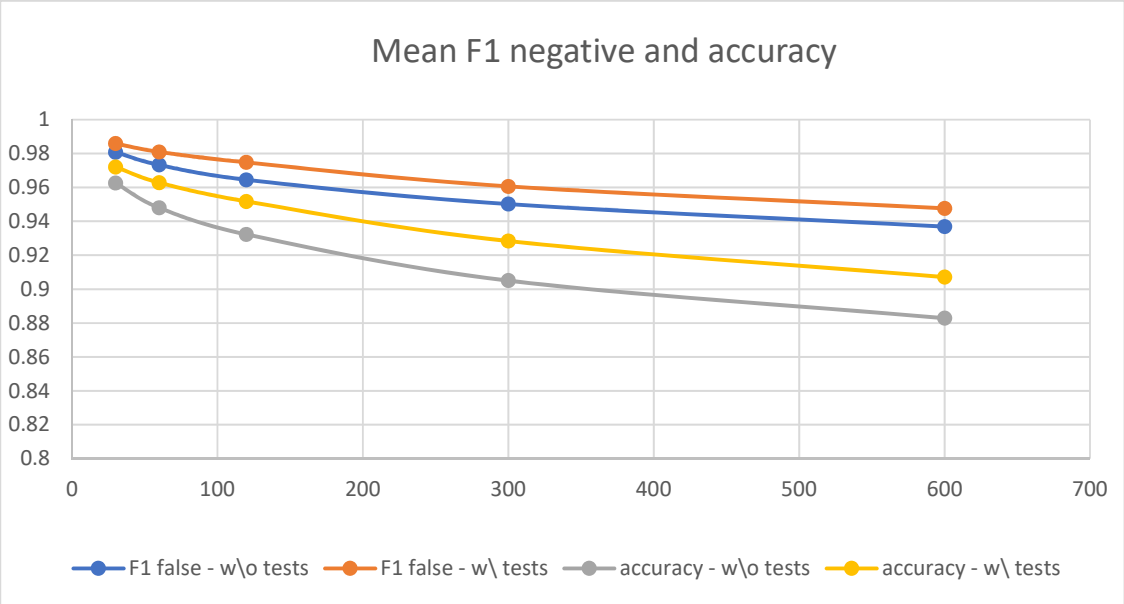

Figure 5.9: Mean F1 scores for negative cases and accuracy without oversampling

by the fact, that larger intervals provided better results.
In order to address this issue, we conducted experiments with oversampling. The results from these tests were decent, the accuracy was around 90%. Whether the model built from this approach is applicable to live task switch detection is questionable, however. The false positive frequency seems problematic.
We tried to improve the results further by using a different threshold and came to the conclusion, that changing the minimum confidence needed in order to identify a task switch did not affect the results.
In our last experiment, we tried to identify impactful features. There were some tendencies for tests, file savings, and active time. However, they were not consistent.
Additionally, we examined the influence of test presence. Since many developers did not test at all, we wanted to show the difference in accuracy. Absolutely, the differences were small while relatively, having tests almost doubled the precision.

5.5 Task Boundaries within Sessions

Software development requires numerous activities besides coding. Requirements have to be understood, tasks have to be clarified. Consequently, developers are not working in their IDEs all the time. Meetings and similar activities fragment the workday into many coding sessions. For each event, FeedBaG assigns the corresponding session ID. These sessions are limited by the IDE opening and closing actions. In this evaluation, we tried to build a model based on the interaction data of sessions. Compared to the previous two experiments, the data was even more granular. We already established that additional knowledge might influence the developer behaviour. If the results are significantly better than those of the developer focused experiment, we can conclude that developers do not necessarily have an individual work style. Rather developers adapt their style to the circumstances or they do not have any habits at all.

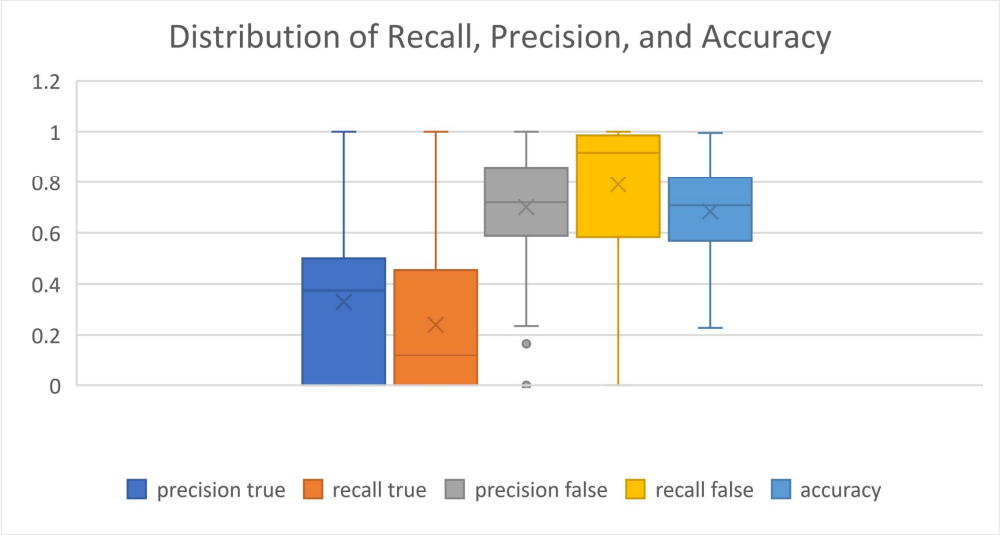

Figure 5.10: Accuracy of Session-based Model Building

Approach The developer-based experiment showed the most promising results for an interval of five minutes. For this experiment, we examined the qualitative differences between the developer-based and the session-based approach for this interval. The basic process stayed the same compared to the previous two approaches. During the preprocessing step however, there was an additional stage. The data sets had to be split from developers to the more granular sessions.

Results The results of the two previous experiments had a common denominator. While the F1 score for negative cases was very high, the score for positive cases were extremely low. The session based approach however provided different results. Figure 5.10 shows the distribution of precision and recall for both positive and negative cases. Compared to figure 3.4, the distribution was more centred. In other words recall and precision of positive cases were higher while the respective values for negative cases were lower. There were even numerous cases where all task switches were correctly recognized within a session.

In total we analysed 615 sessions from 9 developers. 113 of these sessions contained a task switch. The mean accuracy over all sessions was around 68%. The mean F1 value was around 25%, thus way higher than the respective value of the developer based approach without oversampling. However, the amount of false positives was way increased perceptibly. Consequently, the accuracy was almost 20% lower. We already explained that false positives are way worse than false negatives. A session based model is therefore equally useless as compared to a global or developer specific model.

Nonetheless, we were able to draw some conclusions from these scores. Figure 5.11 shows the distribution of the accuracy for individual sessions of an exemplary developer. The values are spread out from 20% to 90%. While there are certain tendencies around 50% accuracy for this developer, the spread of the accuracy values indicates that developers do not always behave the same. Otherwise, the accuracy would be more concentrated. This claim was additionally supported by the findings of the previous experiment, which had impressively shown that developer behaviour changes over time such that an accurate model was not achievable.

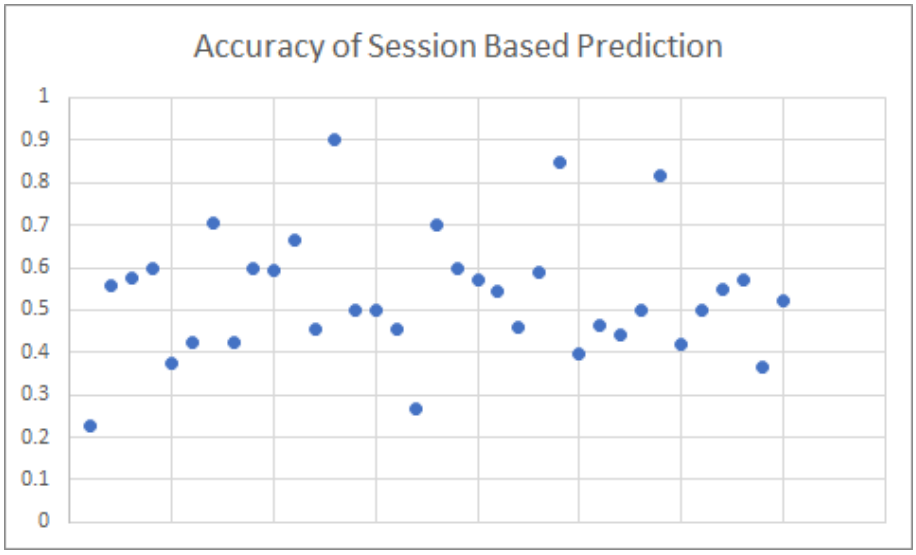

Figure 5.11: Accuracy obtained from Session Based Approach

In conclusion the session based approach provided the best F1 score for task switches of all three tested approaches. However, the overall accuracy of the model suffers heavily. False positives occur more often compared to the developer-specific and global model. Therefore, a session based approach is not applicable either.

Summary

We have analysed several methods and statistics regarding task boundary detection at developer level. We have tested several features which supposedly influence the state of the task. The models gained from periodically extracting the feature vectors, have atrocious precision and recall values for task switches. We tried to improve the results by balancing out the data set as it is suggested in rare data research. The *Resample* technique has provided satisfying F1 scores for both task switches and non-task switches. However, whether this approach is suitable for live task boundary detection from event streams must be evaluated first.
Additionally, we examined whether certain features have bigger influence on the outcome of the logistic regression model than others. While there are certain tendencies, there was no indicator that a specific feature constantly has a positive or negative influence. The weights of the feature show great discrepancy between developers. Therefore, it can be concluded that the way developers tackle tasks is highly individual.

Chapter 6

Discussion

Several experiments were conducted in order to evaluate our proposed approach to task boundary detection in low-level interaction logs. We tested different granularities, reaching from a global model down to a session-based one. While the focus was on detecting task switches for data sets provided by developers, we have also conducted experiments regarding a global model and sessions. For every approach, the same basic algorithm, which was introduced in chapter 3, was used. The experiments mostly differed in the preprocessing step where the data was prepared for logistic regression.

Global Model We conducted a small experiment which aimed to find proof for the possibility of a global mode, i.e. a model which is applicable to the whole data set. The model was trained based on the interaction records of 13 developers and tested against the data set of the three remaining developers. The experiments were conducted with five different sampling intervals: 30 seconds, one minute, two, five, and ten minutes. The model was only able to predict two task switches out of several hundreds.
This strongly indicates, that it is not possible to build a global model. We can draw several conclusions from this. First and foremost, developers behave differently. While there certainly are some tendencies, e.g. that testing is an integral part for some developers, we were not able to find features which strongly indicate an upcoming task switch for the majority of the developers. In fact, the results were so poor, that a global model, solely based on low-level interaction logs, seems highly unlikely.
Compared to the other experiments, the global model provided the worst results. As we have seen before and explain further below, developers seldom have a completely consistent work flow. Even less so is it possible to detect common patterns in interaction logs of multiple developers. After all, software development seems to be approached by individual preferences. Multiple studies have already shown that the type influences how developers complete their tasks. Additionally, based on low-level interaction records, it is not possible to examine what types of tasks or even projects they work on. It is not unlikely that developers behave differently when they have to develop a heavily GUI focused application compared to a functional library. The evaluated developers are volunteers. Therefore, there is most likely a wide disparity between developers in regards of knowledge and experience as well as field of work. We already established, that at least one half of all data sets belongs to professional developers. Usually, they are part of a project team, which ideally sets standards for their tasks. These standards actively influence what steps a developer has to conduct in order to complete a task.
Most developers, who are providing their data to FeedBaG, do not work on the same projects and thus have different DoDs. However, it would be very interesting to conduct experiments which focus on project teams. It seems not unlikely that a global pattern could be discovered within a project team.

With the chosen approach however, it was not possible to obtain a global model. The results were actually so far off from reliably predict task switches, that our low-level actions focused approach does not provide a realistic methodology for this use case.

Developer-based model In our initial project definition, we explained that knowing the developer's context of work can provide valuable information. Based on this data, reasoning about the event stream can be enabled. Therefore, we heavily focused on finding a model for individual developers. FeedBaG provides data that is already split into several sets belonging to one developer each. Due to this characteristic, discovering and testing interaction patterns in developer-specific data was straight-forward.
Unfortunately, we obtained very poor results. We have undersampled negative cases, i.e. non-task-switches, using a periodic sampling. We were able to show that the class imbalance in regard to task switches is a big problem when analysing the stream data. Longer sampling intervals have generally provided better results. The tests were conducted on all 16 data sets which were chosen for this evaluation. We have analysed the F1 scores for both negative and positive cases. While we were able to reliably detect negative cases with a F1 score of at least 0.96, the respective value for positive cases remained below 0.2. In short, the data set suffers from a classic problem in rare event data sets: class imbalance.
Naturally, machine learning on highly uneven class value distribution results in high precision and recall for the majority class, while highly neglecting the minority value. This is due to overshadowing. In fact, an algorithm usually provides the best results for highly imbalanced sets by just always returning the majority class as result for every feature vector.
This problem was addressed with a proven approach called oversampling. We used two different algorithms for this purpose, namely *Resample* and *ClassBalancer*. *Resample* has provided the best results. The mean of all F1 scores of both positive and negative cases was above 0.9 which is very respectable. However, due to the increased precision and recall for task switches, the negative cases suffered heavily. While an F1 score of 0.85 and precision and recall values of more than 0.9 sound respectable, it has to be put in place properly to understand the consequences of this reduction. A developer works around eight hours per day. If a five minute interval is used, this means at most 96 separate intervals. With an accuracy of 85%, an interval would be wrongfully declared as task switch roughly fourteen times per day or almost twice an hour. In reality, developers do not switch the programming task nearly as often. This shows impressively how bad the influence of oversampling for our use case actually is. A good system would provide false positives once per day at most.
Additionally, we have examined the influence of each feature on the outcome of the model. Surprisingly, there were only few tendencies recognisable. It seems like test success has the biggest impact on the model. An unsuccessful test was also the only attribute which strongly indicates that no task switch is upcoming. Other than that, the values were all over the place. The only value, that was not dominated by statistical outliers, was the active time. The results presented in figure 5.7 suggest that developers are rather less active within the IDE during the time leading up to a task switch. This is not a big surprise. The time preceding a task switch usually is not a programming dictated phase. As the previous findings indicate, the last actions before a task is finished are not of programming nature. Tests and builds are run, the workspace is cleaned up, administrative tasks are conducted. All these actions are reflected less than programming in interaction logs.
Also, there was a small tendency for file savings. While there are a lot of outliers, they all were in the negative spectrum. Concretely, this means that for each additional document save, the likelihood of a task switch decreases within the interval. This directly relates to the findings concerning the active time. When there is little programming conducted, there is also not much incentive for saving files.

While an additional oversampling of the minority class can greatly increase the F1 score of task switches, it also introduces countless false positives. This impact is grave enough to reduce the practicality of our approach below satisfying values. Additionally, we were able to show that certain features have a tendency to indicate task switches. Unsuccessful tests heavily decrease the probability of a task switch. Unsuccessful builds, high activity and a lot of file savings indicate slightly that a task switch is not immediate. All other features, such as file closings, number of events in the interval and the time since the last build were not able to consistently indicate whether a task switch is nearing or not. It is therefore not possible to make a generally applicable statement on how they influence the probability of task switches.

Session-based model We have seen great improvement when moving from the global model to a developer specific one. We also theorized that there are many circumstances that influence a developer's behaviour such as knowledge and the type of task they are working on. Consequently, we observed that moving from a developer level to a session level, the results were improved. There were numerous sessions, in which all task switches were correctly recognized. On the other hand, the mean accuracy was only 70%. This value is way lower than those from the previous experiments. Besides the low accuracy, this approach lacks in practicality. Sessions do not necessarily contain task switches. Additionally, they are often to short to actually build a model based on the interaction data.

6.1 Threats to Validity

The results obtained by our approach are unexpectedly poor. However, we have used a proven algorithm from a different but similar problem domain. This section will take a closer look on the soundness of our approach and evaluation before discussing whether these results have any consequences on real world applications.

Internal Validity

We provided extensive explanations how the problem was approached and how the approach was evaluated. However, there might be some shortcomings in our approach and evaluation. This section will provide an overview of possible problems with the evaluation process.

Commits as Task Switches In chapter 2.3 we defined the principle of a task in software development. However, we did not have any concrete meta data considering task switches in FeedBaG's interaction records. We assumed that commits happen at the end of tasks and thus can be used as indicators for task switches.
It is not proven that this assessment is correct, though. While both, tasks and commits, should concern a package of related work, this is not necessarily the case. While best practices suggest using a single commit per task, this is only a best practice and not a written law. Developers might, based on their task's scope, have multiple commits per task. This would strongly influence the significance of the features used in the experiments. When strictly following test-driven development, a possible scenario might require a commit after writing the tests. This would automatically mean that there was neither a successful test nor a build before the commit. Therefore, the results would be inherently different when comparing the point at which the tests were committed with the actual task switch.

Feature Choice Section 3.2.1 has shown how it is possible to extract features from raw interaction logs. We have also provided reasons for the choice of features. However, software development involves a lot of actions. Therefore, it is likely that there are features we have not taken into consideration but influence the build model. Present research has mainly focused on the methods edited by the developer [KF17, CS08]. This information is not necessarily provided by low level-level interaction logs. The enriched event stream provided by FeedBaG however contains Simplified Syntax Trees (SSTs) which provide snapshots of source code. We have not exploited these structures. It might also be possible to find correlations between the edited source files and task switches. There is a simple idea behind this assumption. It is likely that the same files are edited during a task. Once the task is switched, different files are targeted by the developer. This is also an approach which is used for task boundary detection in computer related work. However, this isn't trivially applicable to our approach as a feature can't be extracted easily. A possible way to take files into consideration with logistic regression is to calculate the files overlapping two successive intervals.
The file related feature is only one example of many possible features we have not touched as part of our work. In order to theorize additional features, the developer behaviour has to be studied intensively. Nonetheless, the feature could have been badly chosen. By using better features, it might also be possible to obtain better results.

Data Set Initially, we obtained a data set consisting of interaction logs from 82 developers. Unfortunately and surprisingly, almost one half of the data sets did not contain any commit related events. This does not necessarily mean that these developers do not use any version control. FeedBaG only records interactions with GIT while there are other version control systems used such as SVN and Mercurial. Nonetheless, the size of the data set had to be decreased by large margin.
Additionally, we decided to remove all data sets which did not contain test runs initially. To our surprise, only 16 developers used both, GIT and tests. This ratio does not seem very realistic. On the other hand, not all users of FeedBaG are professionals. It is not impossible for hobby developers as well as hobby projects to enjoy less strict project management. Removing interaction logs which do not contain version control events was a necessity. However, excluding data sets without test runs strongly influences the outcome.
This poses a big threat to the integrity. It might be possible that a different data set provides different results. Presumably interaction logs from professional developers, which use the tool in their daily work, follow tighter bounds in order to complete tasks.

External Validity

Opposed to previous research in this field, we did not rely on a laboratory setting. Zou and Godfrey's evaluation of Coman's algorithm is a great example for showing how results gained in a isolated can differ heavily from those gained in a real-life setting [ZG12]. Our approach also does not suffer from the well-known Hawthorne effect. Developers are not actively observed. Additionally, the approach we provided is applicable to almost every data set as long as it is chronologically ordered. It does not even necessarily rely on contextual information. However, the meaningfulness of features is increased heavily by such information.
On the other hand, the data set has some unexpected characteristics. We did not expect that only roughly 20% of all developers make use of both, GIT and testing. These were to interactions our implementation relied heavily. One interpretation of this might be that developers mainly upload data from their private projects and that they follow less strict guidelines. Or maybe the work flow is not embedded as much into the definition of done as expected, therefore also professional work would not provide recurring patterns. However, this is just speculation and we can not say for

sure that the results are that bad for all data sets. In order to make more precise statements on this issue, the approach must be cross validated against other data sets.

6.2 Future Work

While the results were worse than expected, we are convinced that our approach can be successful. Our experiments were based on a finite set of features that were extracted based on the event stream. Given more meaningful features, we are sure that our approach can provide a developer specific model which can reliably predict task switches.
One of the biggest drawbacks of our approach was that we were not able to use commits in order to predict task switches. Since FeedBaG does not have meta data concerning tasks, we intend to enrich the data stream with task related meta information. Supposedly, being able to use commits as feature vectors instead of as label would increase the accuracy of our approach noticeably.
One major difference to previous work on this subject was the focus on low-level events instead of the modified code. Since our approach didn't lead to convincing results, it should be evaluated how these two approaches can be combined in a single one. FeedBaG already provides a hook for these kind of operations in SSTs. This additional source code information should definitely be exploited more.
On the other hand, the proposed algorithm might not be precise enough for task boundary detection at all. Therefore, the context layer of FeedBaG's data might be abusable for a different approach. Research in the field of task detection often suggests observing the modified artefacts. The problem of recognising tasks could be transformed into a graph problem which deals with finding connected sets of source code files. The basic idea behind such an approach would be to group the available source files into related sets. Once the focus shifts from related files to completely unrelated ones, a task switch might be indicated. This concept should be tested separately, however.

Chapter 7

Summary

In this thesis, we conducted research on whether task switches can be detected in raw in-IDE interaction logs. We adapted the pattern discovery algorithm used in web mining in order to predict task switches based on low-level interactions.
Based on the assumption that the completion of tasks consists of several, recurring steps which can be observed in interaction records, we applied an interval based sampling to a data stream. At each sampling step, the tasks state was extracted. We applied logistic regression to these sampling points in order to build a model which predicts task switches.
This method was applied in three different ways. First, we found out that a global model for our data set does not exist. The model built from 13 developers was not able to predict task switches in the test data set consisting of data from 3 developers. We concluded from these results, that task completion is highly dependant on the developer as well as the project they are working on. Thereafter, we found out that a naive approach on a developer level can not reliably predict task switches. Due to the infrequency of task switches, the data set was highly skewed in favour of non-task switches. To deal with this issue, we applied two oversampling techniques which balance out the data set. Using these techniques, we were able to obtain accuracies from over 90%. However, the frequency of false positives became too high for this approach to provide reliable information for researchers.
In a further step, we wanted to analyse whether developers behave consistently within a session. The results from this experiment showed, that a session based approach is neither practical nor precis.

In conclusion, our approach for detecting task switches in low-level IDE interaction log was not successful. We assume however, with better features, it is possible to predict task switches based on our technique. Therefore, automated task boundary detection based on low-level IDE interaction data seems feasible.

Bibliography

[APN16] S. Amann, S. Proksch, and S. Nadi. Feedbag: An interaction tracker for visual studio. In *2016 IEEE 24th International Conference on Program Comprehension (ICPC)*, pages 1–3, May 2016.

[APNM16] S. Amann, S. Proksch, S. Nadi, and M. Mezini. A study of visual studio usage in practice. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 1, pages 124–134, March 2016.

[AWS06] Richard Atterer, Monika Wnuk, and Albrecht Schmidt. Knowing the user's every move: User activity tracking for website usability evaluation and implicit interaction. In *Proceedings of the 15th International Conference on World Wide Web*, WWW '06, pages 203–212, New York, NY, USA, 2006. ACM.

[BBVB+01] Kent Beck, Mike Beedle, Arie Van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, et al. Manifesto for agile software development. 2001.

[BMM17] Mateusz Buda, Atsuto Maki, and Maciej A. Mazurowski. A systematic study of the class imbalance problem in convolutional neural networks. *CoRR*, abs/1710.05381, 2017.

[BÖ14] Yalin Baştanlar and Mustafa Özuysal. *Introduction to Machine Learning*, pages 105–128. Humana Press, Totowa, NJ, 2014.

[CA18] J.D. Clark and I. Asimov. *Ignition!: An Informal History of Liquid Rocket Propellants*. Rutgers University Press, 2018.

[CGS00] Filip Coenen Gilbert. Swinnen. A framework for self adaptive websites: Tactical versus strategic changes. 08 2000.

[Cla] Classbalancer. http://weka.sourceforge.net/doc.dev/weka/filters/supervised/instance/ClassBa [accessed 26. January 2018].

[CMS97] R. Cooley, B. Mobasher, and J. Srivastava. Web mining: information and pattern discovery on the world wide web. In *Proceedings Ninth IEEE International Conference on Tools with Artificial Intelligence*, pages 558–567, Nov 1997.

[Coh05] Mike Cohn. *Agile estimating and planning*. Pearson Education, 2005.

[CS08] I. D. Coman and A. Sillitti. Automated identification of tasks in development sessions. In *2008 16th IEEE International Conference on Program Comprehension*, pages 212–217, June 2008.

[Data] Data integration approaches | howstuffworks. https://computer.howstuffworks.com/data-integration2.htm. [accessed 26. January 2018].

[Datb] Data preprocessing & data wrangling in machine learning | deep learning - xenonstack.

[Dav13] N. Davis. Driving quality improvement and reducing technical debt with the definition of done. In *2013 Agile Conference*, pages 164–168, Aug 2013.

[DRL12] Didier Devaurs, Andreas Rath, and Stefanie Lindstaedt. Exploiting the user interaction context for automatic task detection. 26:58–80, 02 2012.

[FBH02] David Franklin, Jay Budzik, and Kristian Hammond. Plan-based interfaces: Keeping track of user tasks and acting to cooperate. In *Proceedings of the 7th International Conference on Intelligent User Interfaces*, IUI '02, pages 79–86, New York, NY, USA, 2002. ACM.

[GH11] Des Greer and Yann Hamon. Agile software development. *Software: Practice and Experience*, 41(9):943–944, 2011.

[Gha04] Zoubin Ghahramani. Unsupervised learning. In *Advanced lectures on machine learning*, pages 72–112. Springer, 2004.

[Gil] Navdeep Singh Gill. Log analytics with deep learning and machine learning - hiring | upwork. https://www.upwork.com/hiring/for-clients/log-analytics-deep-learning-machine-learning/. [accessed 26. January 2018].

[Haw] Hawthorne effect. https://www.investopedia.com/terms/h/hawthorne-effect.asp. [accessed 26. January 2018].

[HDW94] G. Holmes, A. Donkin, and I. H. Witten. Weka: a machine learning workbench. In *Intelligent Information Systems,1994. Proceedings of the 1994 Second Australian and New Zealand Conference on*, pages 357–361, Nov 1994.

[HFH+09] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H. Witten. The weka data mining software: An update. *SIGKDD Explor. Newsl.*, 11(1):10–18, November 2009.

[HG09] H. He and E. A. Garcia. Learning from imbalanced data. *IEEE Transactions on Knowledge and Data Engineering*, 21(9):1263–1284, Sept 2009.

[IV06] Renáta Iváncsy and Istvan Vajk. Frequent pattern mining in web log data. 3, 01 2006.

[KB00] Raymond Kosala and Hendrik Blockeel. Web mining research: A survey. *SIGKDD Explor. Newsl.*, 2(1):1–15, June 2000.

[KF17] K. Kevic and T. Fritz. Towards activity-aware tool support for change tasks. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 171–182, Sept 2017.

[KM05] Mik Kersten and Gail C. Murphy. Mylar: A degree-of-interest model for ides. In *Proceedings of the 4th International Conference on Aspect-oriented Software Development*, AOSD '05, pages 159–168, New York, NY, USA, 2005. ACM.

[KMM+07] Y. Kamei, A. Monden, S. Matsumoto, T. Kakimoto, and K. i. Matsumoto. The effects of over and under sampling on fault-prone module detection. In *First International Symposium on Empirical Software Engineering and Measurement (ESEM 2007)*, pages 196–204, Sept 2007.

[LM98] Huan Liu and Hiroshi Motoda. *Feature extraction, construction and selection: A data mining perspective*, volume 453. Springer Science & Business Media, 1998.

[Mav] Maven repository: nz.ac.waikato.cms.weka » weka-dev. https://mvnrepository.com/artifact/nz.ac.waikato.cms.weka/weka-dev. [Accessed on 14. January 2018].

[MBM+17] A. N. Meyer, L. E. Barton, G. C. Murphy, T. Zimmermann, and T. Fritz. The work life of developers: Activities, switches and perceived productivity. *IEEE Transactions on Software Engineering*, 43(12):1178–1193, Dec 2017.

[MFPSS96] Usama M. Fayyad, Gregory Piatetsky-Shapiro, and Padhraic Smyth. From data mining to knowledge discovery in databases. 17:37–54, 03 1996.

[Min14] R. Minelli. Towards self-adaptive ides. In *2014 IEEE International Conference on Software Maintenance and Evolution*, pages 666–666, Sept 2014.

[MKF06] G. C. Murphy, M. Kersten, and L. Findlater. How are java software developers using the elipse ide? *IEEE Software*, 23(4):76–83, July 2006.

[MML15] R. Minelli, A. Mocci, and M. Lanza. I know what you did last summer - an investigation of how developers spend their time. In *2015 IEEE 23rd International Conference on Program Comprehension*, pages 25–35, May 2015.

[MMLB14] R. Minelli, A. Mocci, M. Lanza, and L. Baracchi. Visualizing developer interactions. In *2014 Second IEEE Working Conference on Software Visualization*, pages 147–156, Sept 2014.

[MRT12] Mehryar Mohri, Afshin Rostamizadeh, and Ameet Talwalkar. *Foundations of machine learning*. MIT press, 2012.

[PAN18] S. Proksch, S. Amann, and S. Nadi. Enriched event streams: A general dataset for empirical studies on in-ide activities of software developers. *Proceedings of International Conference on Mining Software Repositories*, page tbd, May 2018.

[PLI02] Chao-Ying Joanne Peng, Kuk Lida Lee, and Gary M. Ingersoll. An introduction to logistic regression analysis and reporting. *The Journal of Educational Research*, 96(1):3–14, 2002.

[PNAM17] Sebastian Proksch, Sarah Nadi, Sven Amann, and Mira Mezini. Enriching in-ide process information with fine-grained source code history. In *International Conference on Software Analysis, Evolution, and Reengineering*, Februar 2017.

[Pro17] Sebastian Proksch. *Enriched Event Streams: A General Platform For Empirical Studies On In-IDE Activities Of Software Developers*. PhD thesis, Technische Universität, Darmstadt, Mai 2017.

[PSV94] D. E. Perry, N. A. Staudenmayer, and L. G. Votta. People, organizations, and process improvement. *IEEE Software*, 11(4):36–45, July 1994.

[RD00] Erhard Rahm and Hong Hai Do. Data cleaning: Problems and current approaches. *IEEE Data Eng. Bull.*, 23(4):3–13, 2000.

[Res] Resample. http://weka.sourceforge.net/doc.dev/weka/filters/supervised/instance/Resample.html. [accessed 26. January 2018].

[RSR09] K R. Suneetha and Krishnamoorthi R. Identifying user behavior by analyzing web server access log file. 9, 01 2009.

[Sam59] A. L. Samuel. Some studies in machine learning using the game of checkers. *IBM Journal of Research and Development*, 3(3):210–229, July 1959.

[SBD+05] Simone Stumpf, Xinlong Bao, Anton Dragunov, Thomas G. Dietterich, Jon Herlocker, Lida Li, and Jianqiang Shen. Predicting user tasks: I know what you're doing. In *In 20 th National Conference on Artificial Intelligence (AAAI-05), Workshop on Human Comprehensible Machine Learning*. Press, 2005.

[SCDT00] Jaideep Srivastava, Robert Cooley, Mukund Deshpande, and Pang-Ning Tan. Web usage mining: Discovery and applications of usage patterns from web data. *SIGKDD Explor. Newsl.*, 1(2):12–23, January 2000.

[SLDH06] Jianqiang Shen, Lida Li, Thomas G. Dietterich, and Jonathan L. Herlocker. A hybrid learning system for recognizing user tasks from desktop activities and email messages. In *Proceedings of the 11th International Conference on Intelligent User Interfaces*, IUI '06, pages 86–92, New York, NY, USA, 2006. ACM.

[SOSt06] Greg Smith, Nuria Oliver, Arun C. Surendran, and chintan thakkar. Swish: Semantic analysis of window titles and switching history. Technical report, March 2006.

[SS97] J. Sola and J. Sevilla. Importance of input data normalization for the application of neural networks to complex industrial problems. *IEEE Transactions on Nuclear Science*, 44(3):1464–1468, Jun 1997.

[Sta] Stack overflow developer survey 2017. https://insights.stackoverflow.com/survey/2017. [Accessed on 16. January 2018].

[Sza07] Victor Szalvay. Glossary of scrum terms - scrum alliance. https://www.scrumalliance.org/community/articles/2007/march/glossary-of-scrum-terms1128, 3 2007. [Accessed on 13. January 2018.

[Ver] Version control best practices. https://www.git-tower.com/learn/git/ebook/en/command-line/appendix/best-practices. [Accessed on 30. January 2018].

[Whaa] What is data aggregation? - definition from whatis.com. http://searchsqlserver.techtarget.com/definition/data-aggregation. [accessed 26. January 2018].

[Whab] What is data generalization | online learning. http://www.learn.geekinterview.com/data-warehouse/data-quality/what-is-data-generalization.html. [accessed 26. January 2018].

[YR11] A. T. T. Ying and M. P. Robillard. The influence of the task on programmer behaviour. In *2011 IEEE 19th International Conference on Program Comprehension*, pages 31–40, June 2011.

[ZG12] L. Zou and M. W. Godfrey. An industrial case study of coman's automated task detection algorithm: What worked, what didn't, and why. In *2012 28th IEEE International Conference on Software Maintenance (ICSM)*, pages 6–14, Sept 2012.