SOFTWARE TRAJECTORY ANALYSIS: AN EMPIRICALLY BASED METHOD FOR AUTOMATED SOFTWARE PROCESS DISCOVERY

A DISSERTATION SUBMITTED TO THE
GRADUATE DIVISION OF THE
UNIVERSITY OF HAWAI‘I AT MĀNOA
IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

IN

COMPUTER SCIENCE

MAY 2015

By

Pavel Senin

Dissertation Committee:

Philip M. Johnson, Chairperson
Kyungim Baek
Guylaine Poisson
Henri Casanova
Daniel Port

Dedication

I dedicate this dissertation to my family: to Inna for her patience and understanding, to Dad and Mom for instilling the importance of hard work and higher education, and to my kids, Alexander and Daria.

ACKNOWLEDGMENTS

I would like to thank many people without whom this endeavor would not be possible. My adviser, Philip Johnson, was on my side throughout my graduate studies giving me guidance and encouragement that were essential for its completion. Prof. Jessica Lin helped me with the understanding of symbolic discretization. Sergey Malinchik helped me with SAX-VSM development.

I would like to thank my committee members: Prof. Guylaine Poisson, Prof. Kyungim Baek, Prof. Henri Casanova, and Prof. Daniel Port for taking the time to read and evaluate this dissertation.

I thank the Hackystat hackers for the initiation into software process analysis, ideas, and discussions. I am grateful to have received support from the research laboratories where I worked throughout the years. Many people, including Maqsudul Alam, Thomas Brettin, Chris Detter, Patrick Chain, Jean Challacombe, Monica Misra, Jacques Nicolas, Mondher Bouzayen, Christophe Klopp, and Maria-Soledad Goni Urriza have taken an active interest in my research and provided me with much needed encouragement.

Jimmy Saw, Tracey Freitas, Johanne Aubé, Aaron Kagawa, and Robert Brewer deserve a special mention as the senior students I have collaborated with and who have been role models I looked up to.

Finally, I acknowledge the long-suffering and seemingly inexhaustible support of my family. I thank my wife Inna, who has put up with a great deal during the pursuit of my "mad dream" – she deserves much of the credit for the dissertation's completion. And last, but certainly not least, a special thanks to my kids, Alexander and Daria, who put a smile on my face after long days and helped me to put things in perspective.

ABSTRACT

Recurrent behaviors are considered to be the basic building blocks of any human-driven goal-oriented process, reflecting the development of efficient ways for dealing with common tasks based on past performance. Thus, the ability to discover recurrent behaviors is utterly important for a bottom-up systematic study, modeling, and improvement of human-driven processes. In the context of software development, whose ultimate goal is the delivery of software, the ability to recognize recurrent behaviors enables the understanding, formal description, and effective guidance of evolving software processes. While a number of approaches for recurrent behavior discovery and software process modeling and improvement have been previously proposed, they typically built upon on-line intrusive techniques, such as observations and interviewing, therefore expensive, suffering from biases, and unwelcome by software developers.

In this exploratory study, I have developed and tested the idea of software process discovery via off-line analysis of software process artifacts. For this, I have prototyped and evaluated the Software Trajectory Analysis framework, which is built upon the definition of the "software trajectory" data type, that is a temporally ordered sequence of software artifact measurements, and a novel technique for temporal data classification, that enables the discovery and ranking of software trajectory characteristic patterns. By analogy with the notion of trajectory in Physics, which describes a projectile path in metric space, a software trajectory describes the software process and product progression in a space of chosen software metrics, whereas its recurrent structural patterns are related to the recurrent behaviors.

The claim of this dissertation is that (1) it is possible to discover recurrent behaviors off-line via systematic study of software artifacts, (2) the Software Trajectory Analysis framework provides an effective off-line approach to recurrent software process-characteristic behaviors discovery. In addition to the extensive experimental evaluation of a proposed algorithm for time series characteristic pattern discovery, three empirical case studies were carried out to evaluate the claim: two using software artifacts from public software repositories and one using the public dump of a Q&A web site. The results suggest that Software Trajectory Analysis is capable of discovering software process-characteristic recurrent behaviors off-line, though their sensible interpretation is sometimes difficult.

TABLE OF CONTENTS

Acknowledgments . . . . . **iv**

Abstract . . . . . **v**

List of Tables . . . . . **xi**

List of Figures . . . . . **xii**

Glossary **1**

1 Introduction **2**

1.1 Preliminaries 2

1.2 Motivation. Software Crisis. 4

1.3 Classical approaches to software process design and improvement 6

1.4 Free/Libre/Open Source software processes 8

1.4.1 Public software repositories 9

1.5 Systematic approaches for software process research 12

1.5.1 Software measurements 12

1.5.2 Software telemetry 12

1.5.3 Knowledge discovery from time series 13

1.5.4 Research hypothesis 14

1.5.5 Software Trajectory Analysis (STA) 15

1.6 Contributions 17

1.7 Dissertation Outline 20

2 Prior and related work **21**

2.1 Software measurements 23

2.1.1 Software measurement history . . . 23
2.1.2 Software measurement theory . . . 24
2.1.3 Software measurements in STA . . . 25
2.2 Mining Software Repositories . . . 27
2.3 Understanding Public Software Repositories . . . 27
2.3.1 Public software artifacts . . . 29
2.3.1.1 Source code management system . . . 30
2.3.1.2 Defect tracking system . . . 30
2.3.1.3 Developer communications . . . 31
2.3.1.4 Q&A websites . . . 31
2.3.1.5 Metadata . . . 32
2.4 Data assimilation . . . 33
2.5 Relevant MSR research on recurrent behaviors discovery . . . 33
2.5.1 Itemset mining . . . 33
2.5.2 Time series analysis . . . 35
2.6 Summary . . . 36
3 Interpretable Time Series Classification . . . 37
3.1 Introduction . . . 37
3.2 Time Series classification . . . 38
3.3 Prior and related work in TSC . . . 38
3.4 SAX-VSM classification algorithm . . . 40
3.4.1 Preliminaries . . . 40
3.4.2 Symbolic Aggregate approXimation (SAX) . . . 42

3.4.3 Bag of words representation of time series . . . 46
3.4.4 SAX numerosity reduction . . . 46
3.4.5 Vector Space Model (VSM) adaptation . . . 48
3.4.6 SAX-VSM implementation . . . 50
3.4.6.1 Training . . . 50
3.4.6.2 Classification . . . 52
3.5 Parameters optimization . . . 52
3.6 Intuition behind SAX-VSM . . . 54
3.7 SAX-VSM performance evaluation . . . 55
3.7.1 Analysis of the classification accuracy . . . 56
3.7.2 Scalability analysis . . . 58
3.7.2.1 Cylinder-Bell-Funnel (CBF) dataset . . . 58
3.7.2.2 Two patterns dataset . . . 59
3.7.3 Classification scalability . . . 59
3.7.3.1 SAX-VSM training scalability . . . 60
3.7.4 Robustness to noise . . . 62
3.7.5 Interpretable classification . . . 63
3.7.5.1 Heatmap-like visualization . . . 64
3.7.5.2 Gun Point data set . . . 64
3.7.5.3 OSU Leaf data set . . . 66
3.7.5.4 Coffee data set . . . 68
3.7.5.5 Characteristic pattern utility . . . 69
3.8 Clustering . . . 69

3.8.1 Hierarchical clustering . . . 69
3.8.2 k-Means clustering . . . 70
3.9 Conclusions an discussion . . . 71
4 Results . . . 73
4.1 Software Trajectory Analysis system overview . . . 73
4.1.1 Software Trajectory Analysis implementation . . . 74
4.1.1.1 STA is generic . . . 75
4.1.1.2 STA is a two-components system . . . 76
4.1.1.3 STA limitations . . . 78
4.2 STA Pilot studies . . . 79
4.2.1 Feasibility study 1: mining Hackystat software telemetry streams . . . 79
4.2.2 Feasibility study 2: mining public software repositories . . . 81
4.2.2.1 Software release pattern . . . 82
4.2.2.2 Software release pattern discovery with STA . . . 82
4.3 STA 2.0 Case studies . . . 84
4.3.1 Case Study 1: Android OS software release recurrent behaviors discovery . . . 84
4.3.1.1 Android OS dataset . . . 85
4.3.1.2 Study design . . . 86
4.3.1.3 Results . . . 88
4.3.1.4 Discussion . . . 88
4.3.2 Case Study 2: PostgreSQL software maintenance and software release recurrent behaviors discovery . . . 91
4.3.2.1 PostgreSQL dataset . . . 92

4.3.2.2 Study design . . . 93
4.3.2.3 Results . . . 94
4.3.2.4 Discussion . . . 94
4.3.3 Case Study 3: mining user-characteristic behaviors in Stack Overflow data 96
4.3.3.1 StackOverflow data . . . 96
4.3.3.2 Study design . . . 97
4.3.3.3 Results . . . 99
4.3.3.4 Discussion . . . 101
5 Conclusion . . . 103
5.1 Dissertation summary . . . 103
5.2 Research summary . . . 104
5.3 Contributions . . . 104
5.4 Future work . . . 105
Bibliography . . . 106

LIST OF TABLES

3.1 An example of the SAX alphabet breakpoints lookup table. 44

3.2 An example of the MINDIST function lookup table. 45

3.3 The SMART notation. 49

3.4 Description of the datasets used in performance evaluation. 55

3.5 Classification accuracy comparison for state of the art nearest-neighbor, interpretable, and SAX-VSM classifiers. 56

3.6 Classification accuracy comparison for state of the art nearest-neighbor and SAX-VSM classifiers. 57

3.7 Comparison of time series classification algorithms characteristics. 72

4.1 Counts of pre- and post- release trajectories corresponding to the *Deleted LOC* dynamics per author and time interval within Android OS kernel OMAP project. . . . 86

4.2 An excerpt from pre- and post-release class-characteristic pattern vectors obtained by mining the *Deleted LOC* trajectories in Android OS case study. 87

4.3 Accuracy of the LOOCV evaluation for Android OS pre- and post- release behaviors classifiers. 88

4.4 Classification results for the Andriod OS release classifier built using the *Deleted LOC* software trajectory. 89

4.5 Accuracy of the LOOCV evaluation for PostgreSQL software trajectories. 93

4.6 Descriptive statistics for the top StackOverflow users. 97

LIST OF FIGURES

1.1 Torvald's response from the Linux mailing list suggesting that practical reasons, i.e., the "real-life" needs, should be always considered over specifications. 10

1.2 The Software Trajectory Analysis design overview. 16

2.1 An overview of the data flow in Software Trajectory Analysis. 22

2.2 An illustration of the relations between software measurements and key responsibilities in project management from SEI Guidebook. 26

2.3 A detailed overview of the Software Trajectory Analysis data assimilation layer. . . 34

2.4 Figures from the previous study by Hindle et al. confirming the existence of periodicity in software development. 35

3.1 An illustration of the time series symbolic discretization via sliding window. 42

3.2 An illustration of the Symbolic Aggregate Discretization (SAX) algorithm. 43

3.3 An overview of the SAX-VSM algorithm. 51

3.4 An illustration of the DIRECT-driven SAX-VSM parameters optimization for *SyntheticControl* dataset. 52

3.5 An illustration of the numerosity reduction strategy effect on parameters optimization process. 53

3.6 An example of the three classes from CBF dataset. 58

3.7 An example of the four classes from Two Patterns dataset. 60

3.8 Classification accuracy and run time comparison for SAX-VSM and 1NN classifiers on CBF data. 61

3.9 Classification accuracy and run time comparison for SAX-VSM and 1NN classifiers on Two Patterns data. 61

3.10 An illustration of the SAX-VSM class-characteristic pattern vectors size evolution for the CBF dataset with increasingly large training set size. 62

3.11 An illustration of the SAX-VSM class-characteristic pattern vectors size evolution for the Two Patterns dataset with increasingly large training set size. 62

3.12 An illustration of the SAX-VSM classification performance evolution on CBF dataset with added noise and signal loss. 63

3.13 An example of the heatmap-like visualization that exploits SAX-VSM subsequence ranking in order to highlight time series segments that are highly characteristic to the class. 65

3.14 An example of the heatmap-like visualization for Two Patterns dataset. 65

3.15 Best class-characteristic subsequences (right panels, bold lines) discovered by SAX-VSM in the *Gun/Point* data set. 66

3.16 The best class-characteristic subsequences (top panels, bold lines) discovered by SAX-VSM in the *OSULeaf dataset*. 67

3.17 The best class-characteristic subsequences (left panels, bold lines) discovered by SAX-VSM in the *Coffee data set*. 68

3.18 A comparison of the distance metrics performance in hierarchical clustering for the subset of three *SyntheticControl* classes: *Normal, Decreasing trend*, and *Upward shift*. 70

4.1 A schematic overview of the very first and the latest STA implementations. 76

4.2 An example of the STA database schema used in Android OS case study which targets the discovery of recurrent behaviors from a history of software change records. 77

4.3 Results of the pilot STA study. 80

4.4 An example of the discovery of recurrent patterns in software trajectories constructed by measuring Android OS repository source code change artifacts. 83

4.5 The dynamics of *Deleted LOC* measurements throughout Android OS kernel OMAP evolution. 85

4.6 Examples of the recurrent behaviors discovered using *Deleted LOC* software trajectories from Android OS dataset. 89

4.7 PostgreSQL source code change evolution. 92

4.8 Examples of class-characteristic behaviors discovered by SAX-VSM in PostgreSQL Commit Fest experiments. 95

4.9 Examples of class-characteristic behaviors discovered by SAX-VSM in PostgreSQL Software Release experiments. 95

4.10 StackOverflow contributors activity dynamics overview. 97

4.11 A comparison of the activity pattern visualization techniques. 98

4.12 WebLogo visualization of daily activity for top SO users. 99

4.13 WebLogo visualization of weekly activity for top SO users. 100

4.14 Clustering of StackOveflow user behaviors with STA. 101

GLOSSARY

1NN – **One** **N**earest **N**eighbor. A variant of kNN classification where we assign each entity to the class of its closest neighbor.

DIRECT – **DI**viding **RECT**angles algorithm. A general parameters optimization algorithm that is based on the iterative partitioning of the search space into hyper-rectangles [1] [2].

FLOSS – **F**ree/**L**ibre/**O**pen **S**ource **S**oftware. A computer software that can be classified as both free software and open-source software, and is licensed for free use, copying, study, and change. This software model is the opposite to proprietary software, which is typically closed-source and distributed under a restrictive copyright.

PAA – **P**iecewise **A**ggregate **A**pproximation. A technique for reducing the dimensionality of time series by aggregating its segment to their mean values [3].

SAX – **S**ymbolic **A**ggregate appro**X**imation. A technique for symbolic discretization of the continuous signal [4].

SCM system – **S**oftware **C**onfiguration **M**anagement system. A software system which enables tracking and controlling changes in the software [5].

SCRUM – An iterative and incremental agile software development framework for managing product development [6].

TF*IDF – **T**erm **F**requency – **I**nverse **D**ocument **F**requency. A numerical statistic that is intended to reflect how important a word is to a document in a collection of documents [7].

TDD – **T**est **D**riven **D**evelopment. A software development process that relies on the repetition of a very short development cycle: coding of an automated test, producing the minimum amount of code to pass that test, and refactoring the new code to acceptable standards [8].

XP – E**x**treme **P**rogramming. One of several lightweight software development methodologies based on values of simplicity, communication, feedback, courage, and respect [9].

CHAPTER 1
INTRODUCTION

A central issue addressed in this dissertation is the possibility of recurrent behaviors discovery from publicly available software process artifacts.

As recurrent behaviors are considered to be the basic building blocks of any human-driven *goal-oriented* process, which reflect the development of more or less fixed ways of dealing with tasks *based on past performance* [10] [11], then, the ability to discover recurrent behaviors in the context of software development equates to a highly desirable capacity to discover the evolution of characteristic mannerisms in which developers structure their activities – the antecedent features that form high-level software development processes.

I have explored an approach to this problem based on the transformation of software artifact trails into time series by measurements and on the subsequent application of a novel time series classification technique that enables characteristic patterns discovery, which, as I hypothesize, correspond to recurrent behaviors.

This dissertation identifies the challenges of automated discovery of recurrent behaviors, reviews the relevant previous work, proposes and evaluates a novel time series classification algorithm capable of characteristic patterns discovery, and presents the results of an empirical evaluation of the algorithm's applicability to the problem of recurrent behaviors discovery from public software artifacts.

1.1 Preliminaries

Definition 1. A ***Software Process*** defines a way that software development goes. It enumerates resources and artifacts, but most importantly, it defines a set of activities that need to be performed in order to design, to develop, and to maintain software systems.

Examples of such activities include requirements collection and creation of Unified Modeling Language (UML) diagrams, source code writing, system testing, and others. The intent behind

a software process is to provide control over the software development effort by implementing a global strategy and by structuring and coordinating human activities in order to achieve the goal – to deliver a functional software system on time and under budget.

Definition 2. A ***Software Process Model*** is a complete and unambiguous software process description that guarantees a rigorous specification ready to be executed.

Definition 3. A ***Software Repository*** is a storage location from which software system and its complementary and auxiliary information can be retrieved. For open-source projects, repository often provides means for software project management, such as version control system, defect tracking system, and message boards, which is typically referred to as software configuration management (SCM) system.

Definition 4. ***Software Configuration Management*** system, or simply SCM, is a software system which enables tracking and controlling changes in the software. In the research literature concerned with Mining of Software Repositories (MSR) terms "SCM" and "repository" are often used interchangeably as they simply point to the source of the data used for studies.

Definition 5. A ***Software Artifact*** is one of numerous products and byproducts of a software process - a use-case, an UML class diagram, a change record, or a bug report. It is common in Software Engineering to keep software artifacts organized with the help of an SCM system.

Note, that while artifacts play an important role in software processes, where they are used to support software development activities and reused to document the resulting software, they are not created in order to enable a scientific research.

Definition 6. A ***Software Artifact Trail*** is a collection of software process artifacts ordered by the artifact's creation time.

Examples of software artifact trails include a software project's source code change records ordered by commit time and a user's questions at the StackOverflow website ordered by post time.

Definition 7. A ***Software Metric*** is a characteristic of a software or a software process that can be objectively measured.

While I discuss software measurements in detail later in the dissertation, examples of software product metrics include the size of a software system measured in lines of code (LOC) or in function points (FP), and the number of defects discovered in a delivered system. Examples of software process metrics include the velocity of a software process called "churn", that measures the amount of LOC changed per day; the response time to fix an issue; and the "technical debt", that measures deterioration of the code quality over time.

Similarly to other sciences, measurements in Software Engineering are essential for establishing systematic research. Product and process metrics are also important in software project management where they are used in order to derive high-level software project metrics including cost, schedule, and productivity.

1.2 Motivation. Software Crisis.

Contemporary software projects deal with the development of complex software systems and typically have a long life cycle - well over decade. A project's development and maintenance activities are usually carried out by geographically distributed teams and individuals. The development pace, the experience, and the structure of a development team continuously change with project progression and as developers join and leave. When combined with schedule and requirements adjustments, these create numerous difficulties for stakeholders, developers, and users, ultimately affecting the project's success [12].

This software development complexity phenomenon was identified in 1968 as the "Software crisis" [13], and was addressed by bringing the research and the practice of software development under the umbrella of Engineering in an effort to provide the control over the process of software development. Following the Engineering paradigm, numerous methodologies of software design and development processes, known as *Software Processes*, were proposed [14]. Some of these were further formalized into Software Process Models - industrial standards for software development such as CMM [15], ISO [16], PSP [17], and others [18].

In spite of this effort, industrial software development remains error-prone and more than half of all projects ending up failing or being very poorly executed [19]. Some of them are abandoned

due to running over the budget, some are delivered with such low quality, or so late, that they are useless, and some, when delivered, are never used because they do not fulfill requirements.

By the analysis of software project failures, it was acknowledged that the Engineering paradigm may not be an adequate way to control software development processes due to the large discrepancies between problems in Software Engineering and in any other Engineering field [20] [21] [12] [22]. The chief argument supporting this point of view is the drastic difference in the cost model: while in Software Engineering there is almost no cost associated with materials and fabrication, these usually dominate cost in all other Engineering disciplines. Ironically, Software Engineering is suffering from cost and challenges associated with continuous re-design of the product and its design processes – an issue that is hardly seen at all in other Engineering areas. In addition, as it has been shown by numerous studies, engineering-like models of software processes are typically prescriptive and rigid – they are difficult to adapt to the particular organizational structure, to the project specificities, and to changing requirements [23]. Thus, the degree to which an adopted process model structures software processes varies greatly between teams and projects and cannot guarantee success [24] [25]. Finally, an increasing understanding and appreciation of human factors in software development processes over tools, technologies, and standards suggests that human-driven software process aspects are likely to be defining in the software project's fate [26] [27] [28] [23] [29].

However, current alternatives to Engineering-like processes that are flexible, user- and developer-centric, and which often praised for their dynamism, flexibility, and encouragement for innovation – such as Agile and Software craftsmanship – are also affected by the same complexity issues. For example it has been shown that SCRUM (iterative and incremental agile software development framework) does not cover the whole software life-cycle [6], extreme programming (XP) does not scale for large teams [9], and test driven development (TDD) requires an extensive expertise from developers [8]. In addition, the increase in flexibility is often directly linked with increase in uncertainty, creating significant difficulties with project cost and effort estimation [30] [31]. The Free/Libre/Open source software (FLOSS) projects, which are typically less concerned with the cost issues, are also affected by this uncertainty. As it has been shown, most of the open-source

projects never reach a "magic" 1.0 version [32]; among others, the great "infant mortality rate" of open-source projects was related to a burnout, inability to acquire a critical mass of users, loss of leading developer(s), and forking [33].

Currently it is widely acknowledged that there exists no "silver bullet" process which guaranteed to bring a software project to a successful conclusion [34]. Processes are numerous, each has advantages and drawbacks, and each is accompanied with success stories and failure experiences making the process selection difficult and the results of its application unpredictable. This uncertainty, and the alarming rate of software project failures suggest that our understanding of software development "mechanics" is limited and insufficient [22]. The enormous cost of the lost effort, measured in hundreds of billions of US dollars [12] [35] [36], continues to provide motivation for further research on software process design and improvement.

1.3 Classical approaches to software process design and improvement

Traditionally, it has been assumed that software development is performed for a profit in corporate, government, or military settings by people that are mostly collocated together. This assumption shaped early research focused on approaches for on-site "software manufacturing", which were discussed for decades in the Software Engineering literature.

Classical approaches can be divided into two distinct categories. The first category consists of *top-down techniques* which are based on proposing a process that is based on a specific pattern of software development. For example, the Waterfall Model process proposes a sequential pattern in which developers first create a Requirements document, then create a Design, then create an Implementation, and finally develop Tests [37]. Alternatively, the Test Driven Development process proposes an iterative development pattern in which the developer must first write a test case, then write the code to implement that test case, then re-factor the system for maximum clarity and minimal code duplication [8].

While top-down techniques follow the usual path of trial and error, and reflect the creative processes of invention and experimentation, the "invention" of an adequate software process is far from trivial and its evaluation cycle is considerably expensive and long [18] [34]. Moreover, it has been

shown that the process inventors are usually limited in their scope and tend to assume idealized versions of real processes, thus, they often produce "paper lions" - process models which are incomplete, unscientific, and unpredictable [38] therefore likely to be disruptive and unacceptable for end users, at least in their proposed form [39].

The second category of classical software process design and improvement approaches consists of *bottom-up* techniques that focus on knowledge extraction from process event logs for its reconstruction, elicitation, validation, and enhancement [40]. Typically, this task is viewed as a two-levels problem where the process event log is aggregated and transformed into the chain of logical development events at first, and the process model is constructed at the second level [41] [40]. Cook and Wolf, in their pioneering work on software process discovery, have shown the possibility of automated extraction of software process models through the mining of process event logs [42] [43] [44]. Later work by Huo et al. shows the possibility of software process improvement through event logs analysis [45] [46].

The bottom-up approaches, while appearing to be more systematic and potentially less challenging than invention, are also affected by a number of issues, among which observability is the most significant: while live project observations are technically challenging to implement due to the high cost and privacy concerns [40], the post-process data collection, for example through interviewing, significantly affects the process reconstruction validity due to frequent discrepancies between actually performed and reported actions [45]. Yet another significant issue is the insufficient capacity of currently available process discovery and representation techniques to discover and to represent models of distributed and concurrent processes [40].

While distinct in their nature, traditional approaches to software process design and improvement yield similar abstract representations of software processes which are typically expressed formally in a process modeling language or as flowcharts of interconnected software development activities [40] [24]. As process "inventors" put the best of their knowledge, experience, and logical reasoning into the proposed sequence of activities, the process "miners" strive to eliminate the noise and to converge to a concise sequence of activities that is supported by the majority of observations.

This particular attention of traditional approaches to the deterministic and complete model syn-

thesis is often cited as limiting as it assumes idealized and streamlined development environment leaving many variable human factors, such as a team structure, its expertise, work schedule, discipline, and motivation behind – an issue that has been widely recognized [27] [23] [47] [48] but still largely ignored in industrial practices mostly due to the difficulties with human component benefits estimation [49] [50] [51].

1.4 Free/Libre/Open Source software processes

Despite the uncertainty issues discussed above, in recent years we have seen a rise of alternatives to on-site Software Engineering development model – people are coming together over the Internet to create software which they distribute openly, promoting its modification and re-distribution. Surprisingly, they provide very little if any guidance on software processes, effectively allowing any software process to be used as long as it positively contributes to the project's goal. This characteristic freedom of free-software processes, while challenging to traditional schools of Software Engineering and software process research, enables advancements in previously unexplored and underexplored research directions, among which is the role of human factors in software development.

The free-software social movement originates from 1960s and is inspired by the philosophy of source code sharing and its collaborative improvement. The movement was partially formalized in 1983 by Richard Stallman, who launched the GNU Project and founded the Free Software Foundation in 1985. The commonly used term "open-source" was coined later, in 1998 at the very first Open Source Initiative (OSI) meeting [52]. The free-software development community consists of self-organized individuals and teams of mostly non-professional programmers - amateurs, hobbyists, students, and academics. By using the Internet, they collaborate and develop software that is distributed free of charge as source code and is usually called Free/Libre Open Source Software (FLOSS).

Over the years, this software development model has proven its ability to deliver increasingly complex and surprisingly popular software in a truly global scale - when thousands of project's contributors and users are scattered all over the world. A number of FLOSS projects such as Linux and its derivatives, Gnome, Apache HTTP Server, PostgreSQL database, and others, succeeded to

develop and to efficiently manage distributed software processes that are providing control over a large development team and source code base and deliver state of the art software whose quality is similar to or exceeding that of industrial projects [53]. This fact attracted considerable attention not only from industrial companies that seek to emulate successful open source software processes in traditional closed-source commercial environment [54] [55] [56] [57], but also from the software process research community, who wishes to understand the reasons for the success of FLOSS processes [58] [59] [60] [61] [62].

A number of studies conducted on open source processes discovered that they are significantly different from traditional software development at many levels. In particular, the flexibility of open source processes and their inherent capacity to adapt to changing requirements is often cited as the most prominent. Consider an exploratory study performed by Sacchi et al. [62] in which they confirmed that requirements elicitation, analysis, specification, validation, and management of open-source systems are drastically different from traditional approaches where mathematical logic, descriptive schemes, and UML models are usually used. The authors provide numerous evidence that OSS requirements are neither prescriptive nor proscriptive in terms of what should be or what might be done and are instead typically implied simply by discourse of the project participants and, most importantly, by implementation assertions. As yet another example reflecting the importance of software implementation, consider the message posted by L. Torvalds that clearly highlights the preference of practical reasons over specification in Linux kernel development in Figure 1.1.

A lack of explicit specifications, however, creates numerous difficulties in studying open-source processes, as it becomes difficult to understand how the software project got from “here” to “there”. A typical way of uncovering such information is by mining of public software repositories.

1.4.1 Public software repositories

The proliferation of open-source development continues to create publicly available software process artifacts at an increasingly high rate, changing the software process research landscape by providing data covering the full software development life cycle for free. Currently, public code hosting sites such as SourceForge, GoogleCode, and GitHub host thousands of FLOSS projects of-

Re: I request inclusion of SAS Transport Layer and AIC-94xx intothe kernel

From: Linus Torvalds
Date: Thu Sep 29 2005 - 15:03:11 EST

- **Next message:** Dave Jones: "Re: [howto] Kernel hacker's guide to git, updated"
- **Previous message:** Linus Torvalds: "Re: [PATCH] Fix IXP4xx MTD driver no cast warning"
- **In reply to:** Willy Tarreau: "Re: I request inclusion of SAS Transport Layer and AIC-94xx into the kernel"
- **Next in thread:** jerome lacoste: "Re: I request inclusion of SAS Transport Layer and AIC-94xx into the kernel"
- **Messages sorted by:** [date] [thread] [subject] [author]

On Thu, 29 Sep 2005, Arjan van de Ven wrote:
>
> *a spec describes how the hw works... how we do the sw piece is up to us ;)*

How we do the SW is indeed up to us, but I want to step in on your first point.

Again.

A "spec" is close to useless. I have _never_ seen a spec that was both big enough to be useful _and_ accurate.

And I have seen _lots_ of total crap work that was based on specs. It's _the_ single worst way to write software, because it by definition means that the software was written to match theory, not reality.

So there's two MAJOR reasons to avoid specs:

- they're dangerously wrong. Reality is different, and anybody who thinks specs matter over reality should get out of kernel programming NOW. When reality and specs clash, the spec has zero meaning. Zilch. Nada. None.

It's like real science: if you have a theory that doesn't match experiments, it doesn't matter _how_ much you like that theory. It's wrong. You can use it as an approximation, but you MUST keep in mind that it's an approximation.

- specs have an inevitably tendency to try to introduce abstractions levels and wording and documentation policies that make sense for a written spec. Trying to implement actual code off the spec leads to the code looking and working like CRAP.

The classic example of this is the OSI network model protocols. Classic spec-design, which had absolutely _zero_ relevance for the real world. We still talk about the seven layers model, because it's a convenient model for _discussion_, but that has absolutely zero to do with any real-life software engineering. In other words, it's a way to _talk_ about things, not to implement them.

And that's important. Specs are a basis for _talking_about_ things. But they are _not_ a basis for implementing software.

So please don't bother talking about specs. Real standards grow up _despite_ specs, not thanks to them.

Linus

Figure 1.1: Torvald's response from the Linux mailing list suggesting that practical reasons, i.e. the "real-life" needs, should be always considered over specifications. Excerpt from Linux mailing list. http://lkml.indiana.edu/hypermail/linux/kernel/0509.3/1441.html

fering numerous software process artifacts, such as design documents, source codes, bugs and issue records, and developers communications. In addition, Q&A and social websites for developers such as StackOverflow, TopCoder, and others, becoming increasingly popular among software developers and users as places to discuss software issues, to exchange expertise, to learn new tools, and to improve skills.

The scientific community response on the public availability of software process artifacts was overwhelming and a number of venues were established in order to address the increased interest. Since 2004, the International Conference on Software Engineering (ICSE) hosts a Working Conference on Mining Software Repositories (MSR). The original call for papers stated MSR's purpose as *"... to use the data stored in these software repositories to further understanding of software development practices ... [and enable repositories to be] used by researchers to gain empirically based understanding of software development, and by software practitioners to predict and plan various aspects of their project"* [63] [64]. Several other venues including International Conference on Predictive Models in Software Engineering [65], International Conference on Open Source Systems, the Workshop on Public Data about Software Development, and the International Workshop on Emerging Trends in FLOSS Research have also played an important role in shaping and advancing of the new research domain.

Some of the work from this domain addresses the problem of open source software process-related knowledge discovery from artifacts. Probably the most notable and relevant to my research is work by Jensen & Scacchi, where they demonstrated that the knowledge reflecting software processes can be gathered from public systems [58]. In their later work, they showed, that it is possible to reconstruct FLOSS processes by manual mapping of collected process evidence to a pre-defined process meta-model [59] [60]. Another work closely related to my research is by Hindle et al., where they showed that it is possible to discover software process evidence through artifacts partitioning [61], and recurrent behaviors by Fourier analysis of source code change records [66].

However, the work mentioned above and other work based on mining of public software process artifacts show that while public availability of software process artifacts minimizes cost of the observation and eliminates privacy concerns, the nature of public artifacts creates a number of new challenges that limit the scope of the research and significantly elevate its complexity, effectively rendering many of previously developed process research techniques inefficient. For example, the coarse granularity of public software change records hides most of the low-level development activities such as small code edits, unit-test runs, etc., which invalidates the application of many previously developed event-based process mining techniques [61] [67]. Similarly, artifact dupli-

cation due to concurrent and often overlapping processes, as well as the incompleteness of public artifact trails prevent typically deterministic process discovery techniques from producing consistent results [67] [68]. Finally, it was found that the driven by external factors and malleable nature of software development renders state of the art approaches based on time dependent information inefficient [66] [69]. Thus, novel software process analysis and discovery techniques are needed to be developed for public software process artifacts analysis [64].

1.5 Systematic approaches for software process research

In addition to the establishment of an Engineering-like software development paradigm, the acknowledgement of the software crisis led to the development of similar to Engineering project management techniques based on software measurements.

1.5.1 Software measurements

The goal of software measurements is to make objective judgments about software process and product quality. It has been shown that an effective measurements programs help organizations understand their capacities and capabilities, so that they can develop achievable plans for producing and delivering software products. Furthermore, a continuous measurements effort provides an effective foundation for managing process improvement activities, such as CMM [15], PSP [17], [70] [71], ISO 9001 [16], and SPICE [72].

In addition to practical applications, software measurements are extensively used in research – they are the basis of the Empirical Software Engineering research area where researchers base their conclusions on concrete evidence collected through experimentation and measurements of software systems and software processes [73].

1.5.2 Software telemetry

Ideally, by using measurements, a software process and product can be assessed in real-time allowing for efficient in-process decision making. Johnson et al. [74], pioneered this approach by

defining software project telemetry as a particular style of software process and product metrics collection and analysis based on *automated measurements over a specified time interval*. The authors hypothesized that the visualization of multiple streams of collected measurements captures the project and software process state evolution conveying its dynamics to the user. They implemented an in-process software engineering measurement and analysis system called Hackystat [75] that is capable of metrics collection, processing, and telemetry streams visualization. The system's empirical evaluation showed that the visual analysis of multiple telemetry streams aids in in-process decision making, and it is also possible to improve existing software processes by using the knowledge extracted by visual analysis of these streams. At the same time, the authors acknowledged that it is impossible to extract a traditional analytical model that is capable of automating the decision making process and that machine learning application is desirable.

Later, Kou et al. extended Hackystat by implementing the Software Development Stream Analysis Framework (SDSA) that is capable of partitioning telemetry streams into sequences of development "episodes" using pre-defined boundary conditions [76] [77]. By designing "operational definitions" for TDD as sets of specific rules for development episodes, they showed that it is possible to characterize and assign TDD compliance to individual software development episodes. They implemented their approach in Zorro, a software system capable of software process measurement, development episodes inference, categorization, and classification by the TDD conformance. As Zorro is based on pre-defined partitioning and classification rules reflecting our understanding of TDD processes, the authors acknowledged that the application of machine learning techniques may improve systems performance and advance our understanding of software processes.

1.5.3 Knowledge discovery from time series

Both demands for machine learning methods application to the problem of software measurements analysis identified in the previous section can be potentially met by the techniques developed in the research area concerned with unsupervised and semi-supervised knowledge discovery from time series. Time series are typically used as a proxy representing a large variety of real-life phenomena in a wide range of fields including, but not limited to physics, medicine, meteorology, music, motion

capture, image recognition, signal processing, and text mining [78]. While time series usually represent observed phenomena directly by recording their measurable progression in time, pseudo time series are often used for representation of various high-dimensional data by combining data points into ordered sequences. For example in spectrography data values are ordered by the component wavelengths [79], in shape analysis the order is the clockwise walk direction starting from a specific point in the outline [80], in image classification the order is the frequency of pixels sorted by color component values [81].

Many important problems of knowledge discovery from time series reduce to the core task of finding characteristic, likely to be repeated, short sub-sequences that efficiently capture the studied phenomena specificity. In early work these were called as *frequent patterns* [82], *approximate periodic patterns* [83], *primitive shapes* [84], *class prototypes* [85], or *understandable patterns* [86]. Later, similarly to Bioinformatics, these were unified by the term *motif* [87]. Once discovered, time series motifs can be used for research hypothesis generation by their association with known or proposed phenomena [87]. Recent advances in the finding of time series motif and in particular work based on *shapelets* [88] [89] [90] and *bag of patterns* [91] show the great potential of time series motif-based data mining application to almost any phenomena that can be represented as time series.

Since software telemetry streams are in fact time series, that represent the evolution of a software system and a software process measurements in time, their motifs can potentially be discovered and associated with sensible product and process characteristics.

1.5.4 Research hypothesis

In previous sections, I have outlined evidence for the limited performance of traditional Engineering-like software development as well as the the problems encountered by traditional approaches to software process design and improvement when they attempt to take into account the variety of human factors that fall beyond a typical sequence of the development actions. I have identified a number of key differences of FLOSS software development that foster developer- and user-centric processes and which, if systematically studied, can potentially shed light on the role of human-driven aspects

in software development and to improve our overall understanding of software processes. I have pointed out a growing wealth of publicly available software process artifacts that enables systematic FLOSS processes analyses and highlighted the need for novel techniques capable of mining these datasets. Finally, I have explored the possibility of knowledge discovery by time series mining techniques application to software measurements.

All these, along with the results of previous research that has shown that it is possible to discover recurrent behaviors on all levels of software development process hierarchy [17] in industrial [70] and open-source [66] settings, lead to my research hypothesis, that *it is possible to discover the basic blocks of software processes - recurrent behaviors - from public software process artifacts.*

1.5.5 Software Trajectory Analysis (STA)

Following this hypothesis, I have defined Software Trajectory - an abstract representation of software product and process evolution. As the term trajectory is used in Physics for the approximate path that a moving object draws in a physical space, or in Mathematics, where trajectory defined as the reduced in complexity sequence of states of a dynamic system (a Poincaré' map), *Software Trajectory is a curve that only approximately describes the path drawn by an evolving software system or by an ongoing software process in the chosen metric space*. The analytical technique based on software trajectory construction and its analysis I have called Software Trajectory Analysis (STA).

In a preliminary pilot study targeting the possibility of characteristic subsequences discovery from software telemetry streams, I have added an analytical module based on characteristic patterns mining to Hackystat system. This early STA implementation exploited the transformation of real-valued software telemetry streams into short overlapping symbolic sequences with Symbolic Aggregate approXimation (SAX) [92] and their consequent occurrence frequency (i.e., support) -based ranking. While the pilot STA implementation required the user to specify a number of non-intuitive parameters for SAX transform and a threshold for the pattern discrimination, some of the discovered frequent patterns were easily associated with characteristic recurrent software development behaviors, such as consistent effort or frequent testing, and the system performance was found satisfactory [93]. Later, the system was improved by the addition of symbolic motif-mining and visualization

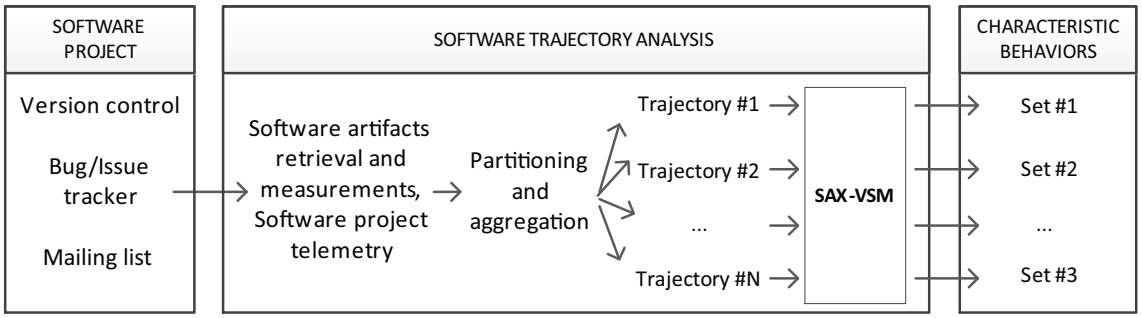


Figure 1.2: The Software Trajectory Analysis design overview. At first, software measurements are acquired directly from an external measurement engine such as Hackystat, or/and by collecting and measuring of software artifacts. Next, the measurements are used by an expert for construction of a set of software trajectories that potentially can shed light on a research question. Finally, recurrent characteristic patterns are discovered and weighted by class importance with SAX-VSM.

algorithms, which not only made the frequent patterns discovery subsystem more efficient, but aided in patterns comprehension through an intuitive visualization.

However, when the system was applied to time series built by measurement of public software artifacts, its performance significantly deteriorated, affected by coarse granularity, poor informational content, noise, and a significant amount of missing values.

Addressing the identified data-mining techniques limitations, I have developed a novel unsupervised technique for time series classification called SAX-VSM, that enables discovery and ranking of class-characteristic patterns, requires no input parameters, and is rotation-invariant and robust to the noise and missing values [94]. In turn, as I shall show later, SAX-VSM -based STA implementation whose overview is shown in Figure 1.2 is capable to discover sensible characteristic subsequences from a wide variety of software process artifacts.

Taking into account all of the above, Software Trajectory Analysis is an automated systematic approach to recurrent behaviors discovery based on software artifact measurements and mining. In contrast with previously proposed systems that were built upon quantitative analyses of atomic development entities such as actions or episodes, or were relying on pre-defined reference process models, STA focuses on the unsupervised discovery of naturally occurring phenomena - recurrent behaviors.

By its design, Software Trajectory Analysis addresses a number of known issues that previously

complicated and limited large scale studies on software processes. First of all, Software Trajectory removes all in-process (real-time) measurement costs and privacy concerns since it relies solely on off-line measurements of public software artifacts. Secondly, STA does not depend on any prior knowledge about software processes or any model - unsupervised data mining techniques, such as SAX-VSM, intended to be used in order to bootstrap knowledge by extracting of data summaries. Finally, STA does not aim at the discovery of complete processes or rigid rules for software development, instead, it yields a set of possible behaviors applicable in a particular situation, i.e. a "point in the software project life cycle" [95].

1.6 Contributions

My contributions include the Software Trajectory Analysis approach (STA) for recurrent behaviors discovery from software process artifacts, the SAX-VSM algorithm for interpretable time series classification that powers-up STA, their implementation, and empirical evaluation:

1. **Software Trajectory Analysis.** The inherent complexity and longevity of software development processes makes their study in real time expensive and challenging, especially at large scale. In addition, the contemporary practices of highly distributed software development, that usually allow the significant variation in software processes, demand new analytical techniques.

   In this work I propose STA – a software process analysis technique that targets the off-line discovery of recurrent behaviors through the systematic analysis of software process artifacts [93] [96]. STA represents a significant contribution to the field of Software Engineering where it can be applied to problems of software process discovery, software process elicitation, software process improvement, and mining of software repositories.

2. **Interpretable time series classification with SAX-VSM.** In order to improve STA performance, I have invented a novel algorithm for interpretable time series classification called SAX-VSM, which I present in this dissertation. SAX-VSM is a general algorithm that represents a significant contribution to the field of time series classification addressing its two core

problems: the characteristic feature selection and the classification results interpretation [94].

SAX-VSM automatically discovers and ranks time series patterns by their class-characteristic power, which not only facilitates time series classification, but provides an interpretable class generalization. These algorithm's strengths are essential for STA performance – they facilitate unsupervised characteristic patterns discovery from software trajectories and convey the understanding of performed software processes by association of the discovered patterns with recurrent behaviors.

3. **Symbolic Aggregate Discretization (SAX) parameters optimization.** SAX-VSM relies on the symbolic discretization procedure (SAX) that requires three parameters to be specified as input. The parameter values are non-intuitive to pick and to the best of my knowledge no solution exists for their optimization. In this work I propose a SAX parameters optimization procedure that guarantees to find the optimal parameters set for SAX-VSM-based time series classification. The approach is based on a general parameters optimization scheme called DIRECT [2] and a common cross-validation-based cost function. The proposed optimization process converges to the optimal parameters set by orders of magnitude faster than a grid-based search [94].

   This general solution for SAX parameters optimization is a significant contribution to the field of time series analysis and classification since it can be adopted for other SAX-based algorithms. In STA this technique aids discovery of the best process-characteristic recurrent behaviors duration and their discretization granularity.

4. **Novel time series class-characteristic subsequence heatmap-like visualization.** SAX-VSM ranks all extracted via sliding window time series subsequences by their class-characteristic potential. By combining the rank coefficients of all subsequences that span a time series point, it is possible to color each point according to its class-characteristic power. This visualization produces a heat map-like time series plot that instantly highlights time series segments according to their class specificity [94].

   In STA this technique enables rendering of many characteristic recurrent behaviors in a single

comprehensive display that aids understanding and interpretation of recurrent behaviors by their association with software trajectory structural features. This technique is a significant contribution to the field of time series data mining and knowledge discovery.

5. **SAX-VSM implementation and empirical evaluation.** SAX-VSM was implemented in Java and open-sourced [97]. I developed SAX-VSM over a period of five years, and it currently consist of over 60,000 lines of source code. This software library represents a significant contribution to the fields of time series analysis, classification, and data mining.

   I have evaluated SAX-VSM classification accuracy, parameters optimization efficiency, and the interpretability of results on a set of 45 classic time series classification problems. The results of evaluation show, that the proposed algorithm is competitive with, or superior to, other techniques in time series classification. At the same time, SAX-VSM is capable to efficiently and effectively discover and rank class-characteristic patterns providing the superior results interpretability and meaningful visualization [94].

6. **STA reference implementation and evaluation.** Software Trajectory Analysis was also implemented in Java and open-sourced [98]. I developed STA over a period of five years and the project is currently consist of over 40,000 lines of source code.

   This dissertation presents the results of STA performance empirical evaluation based on use case studies. Specifically, the case studies-based evaluation shows STA capacity to discover recurrent behaviors including:

   (a) the software release characteristic recurrent behaviors from Android OS and PostgreSQL software development processes;

   (b) the "Commit Fest" characteristic recurrent behaviors from PostgreSQL software development process;

   (c) the characteristic daily and weekly activity patterns of top StackOverflow contributors.

   STA reference implementation and its performance evaluation represent a significant contribution to the field of Software Engineering and to the area of Mining Software Repositories in

particular, as they not only indicate the feasibility of recurrent behaviors discovery, but allow to jump-start new research projects.

1.7 Dissertation Outline

The rest of this dissertation is organized as follows. Chapter 2 discusses related work from software process discovery and software repository mining areas. Chapter 3 discusses relevant work from research areas concerned with time series classification and temporal data mining, and proposes the SAX-VSM algorithm. Chapter 4 shows the Software Trajectory Analysis framework design, explains its implementation, and presents results of its empirical evaluation. Chapter 5 concludes and discusses several directions for future study.

> Design and programming are human activities; forget that and all is lost.
>
> Bjarne Stroustrup

CHAPTER 2
PRIOR AND RELATED WORK

Software Trajectory Analysis (STA) consists of two components: the *software artifacts retrieval and measurement machinery* (i.e., a data assimilation layer), and the *software trajectory characteristic patterns discovery module* (i.e., a data analysis layer). A high-level overview of the information flow through these components is shown in Figure 2.1.

The artifacts retrieval and measurement machinery refers to a way that software artifacts are collected, measured, and enriched with metadata. Currently, STA is capable of retrieving and processing the data from OSS Software Configuration Management system (SCM) components such as version control, defect management, and communications management systems. In addition, STA is able to assimilate data from other data sources among which are community-driven Q&A websites and the Hackystat system [93].

STA is not limited only to these data sources. As public repositories are highly heterogeneous and continuously evolving, STA adopts the Software Repository Mining (MSR) strategy for data assimilation, unification, and off-line enrichment, where public artifacts are retrieved and stored "*as is*" (i.e., mirrored) first, measured second, and enriched with metadata as the final step [99] [100] [101]. Similarly to other systems for mining software repositories, STA relies on a relational database engine for data storage and indexing – this solution not only enables an interactive workflow and a federated access to the data, but allows for effective measurements partitioning and aggregation, which is an *essential capability* for efficient software trajectories construction. Overall, the STA data assimilation layer is designed in a way that conforms to the field's best practices allowing its extension for any data source that is capable of providing data for STA analysis.

The software trajectory characteristic patterns discovery module is an analytical machinery that is responsible for discovery of characteristic recurrent patterns in a set of software trajectories provided as the input. Conceptually, this module can embed *any data mining algorithm* that is capable of discovering recurrent patterns from sequential data, such as one of the numerous algorithms for time series motif discovery [102].

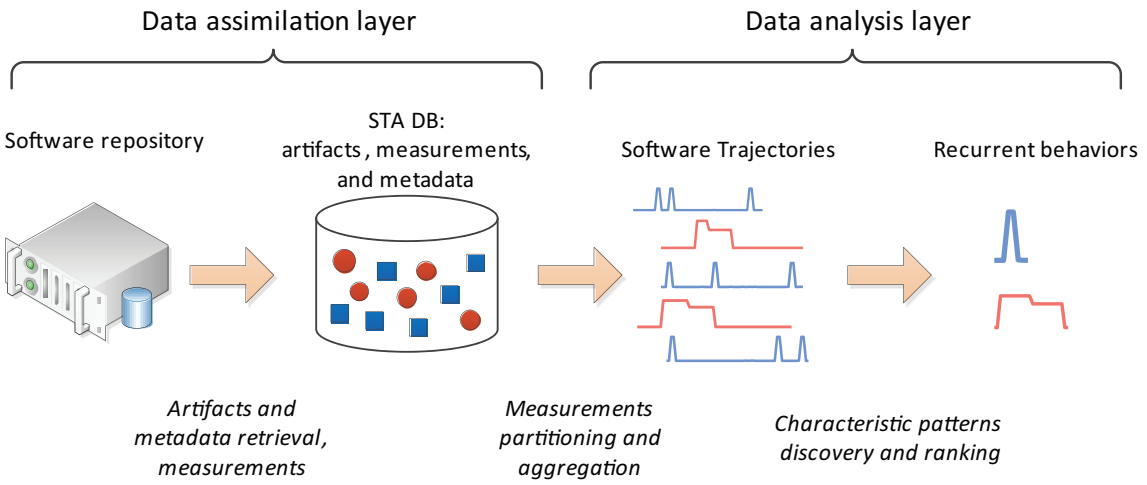


Figure 2.1: An overview of the data flow in Software Trajectory Analysis. Software artifacts are retrieved, enhanced, and measured within the data assimilation layer. Next, based on the user input, classes of software trajectories are constructed. In turn, the data analysis layer performs comparative analyses of software trajectories that yield sets of ranked class-characteristic behaviors. Note, that for the clarity only two classes of trajectories shown whereas STA is capable of discovering class-characteristic patterns from many classes at once.

However, the specificity of software trajectories and the pattern of interest, i.e., recurrent behavior, places a number of constraints that limit the applicability of known algorithms. First of all, the algorithm must be able *to discover recurrent patterns without any prior knowledge about their length, shape, amplitude, and occurrence frequency*, as these are naturally expected to differ between projects, problems, or even subsets of trajectories from the same project. Secondly, it must be capable to *learn from a very small training data set* – the property that has been shown crucial in predictive modeling and knowledge mining from software repositories where data is sparse [103]. And finally, the algorithm must provide an automated mechanism for *patterns ranking according to their relevance* in order to allow their efficient review by human experts since it is impossible to define a pattern "interestingness" or "importance" a priori.

The STA characteristic patterns discovery module implementation relies on SAX-VSM, a novel algorithm for characteristic patterns discovery from time series that I shall propose, describe, and evaluate in the Chapter 3. This algorithm has been designed to address all of the aforementioned requirements.

Later in this chapter, in order to relate Software Trajectory Analysis to other research and to position it among other work, I shall discuss previous work from several research areas. To start, since STA is designed for software measurements analyses, I provide background on software measurements and the evidence of their correlation with software processes. Next, I briefly discuss my earlier exploratory studies conducted with previous STA implementations. Finally, I review research relevant to STA from the Mining Software Repositories (MSR) research field focusing on recurrent behaviors discovery. The work relevant to time series characteristic patterns discovery and SAX-VSM will be discussed in the next Chapter.

2.1 Software measurements

As in all other Engineering fields, measurements are used in Software Engineering in order to establish a systematic approach to software development which provides control over software processes, facilitates their improvement, and, most importantly, makes their result predictable. In addition, software measurements enable scientific research.

2.1.1 Software measurement history

According to Fenton [104], the history of measurements in Software Engineering dates back to the mid-1960's "*...when the Lines of Code metric was used as the basis for measuring the productivity and effort...*", which in fact, predates the establishment of Software Engineering as an independent discipline [13]. Much of the early research concerned with software measurements has been driven by the need for the resource model prediction and forecasting [104], whereas later research has extended towards the problem of software process management [105].

Probably the earliest published work outlining close relations of software measurements and software processes is "Software project forecasting" by DeMillo and Lipton [95] where they point out that software measurements create a basis which allows practitioners and researchers to be "*rational and objective*" about software processes. Remarkably, the authors refer to even earlier notes by Perils, Sayward, and Shaw, who emphasized the role of software measurements in software process management, saying that "*the purpose of software metrics is to provide aids for making optimal*

choice at several points in the life cycle".

With time, the increasing understanding of software measurements objectiveness and their ability to reflect the state of software processes led to the development of measurement-based strategies for software process management and improvement. For example, one of the pioneering strategies for global software process improvement, Total Software Quality Management (TSQM), relies on a set of ten explicitly defined software process and product metrics ranging from the low level product metrics of Lines of Code and Design Complexity to the high-level project management metrics of Schedule and System Testing Progress [106]. Similarly, a local strategy for software process improvement, Personal Software Process (PSP), relies on the broad range of software metrics [107].

In addition to playing an important role in software process management and forecasting, software measurements have become ubiquitous in scientific research. For example in the research field of Empirical (or as it also called Experimental) Software Engineering (ESE), researchers use measurements and experimentation as the basis for research hypotheses generation and their investigation [73].

Recently, due to the proliferation of open source software development and advancements in public software project hosting solutions, a new research area called Mining Software Repositories (MSR) has been established within the ESE field. MSR is specifically concerned with application of analytical techniques to public software repositories [99] [108] [109], thus, the research work from this field is one of the most relevant to my research.

2.1.2 Software measurement theory

In science and in engineering, measurements allow us to formally characterize attributes of an entity by assigning them a numerical, boolean, or symbolic value. The choice of the value type depends on the measurement criteria, such as a dimension, a level, or a degree. Ultimately, the chosen criteria and the scale of used values shall enable an intuitive and precise quantitative comparison between attributes regardless of their qualitative similarity or difference, as it was pointed out by Chapin [110]. In addition, measurement units and scales are usually standardized in order to enable a global comparability.

An entity in Software Engineering can be a physical object, such as a program or a use case diagram, an event, such as a software release, or a software artifact, such as a bug report. A measurable entity's attribute can be its property or a feature, such as the program's size, the amount of defects discovered during testing, or the usability of a software system.

Further, attributes are usually divided into two categories: internal and external. While measures for internal attributes are computed based on the entity itself, external attribute measures depend on both the entity and the environment in which it resides – for example a software system testing time varies depending on the performance of a test server.

Finally, as pointed out by Fenton [111], there are two broad types of measurements: direct and indirect. While direct measurements of an attribute do not depend on any other attributes, indirect measurements involve measurements of one or more other attributes. As an example of a direct measurement, consider the size of a system source code or the time developers spent on project. In contrast, the module defect density (ratio of defects number and the module size), or the requirement stability (ratio of initial requirements and total requirements) are indirect measurements.

2.1.3 Software measurements in STA

Software Trajectory Analysis is designed for analyses of software measurements in order to enable recurrent behaviors discovery. In particular, STA exploits the sequential dependency of consecutive measurements for discovering recurrent characteristic patterns in their dynamics (i.e., structural patterns), which, as I hypothesize, reflect recurrent behaviors.

This approach builds upon previous work that confirmed the feasibility of software processes inference through observations (i.e., measurements) of their effect on software product evolution and indicated the possibility of recurrent behaviors discovery.

As an specific example, confirming the observability of software processes through software product measurements, consider the de-facto industrial standard for software measurements application provided by Software Engineering Institute (SEI) in their guidebook [25]. In particular, the authors focus on the software process execution variability issue that significantly affects the software project's schedule and the resulting software system quality. To address the issue, they propose

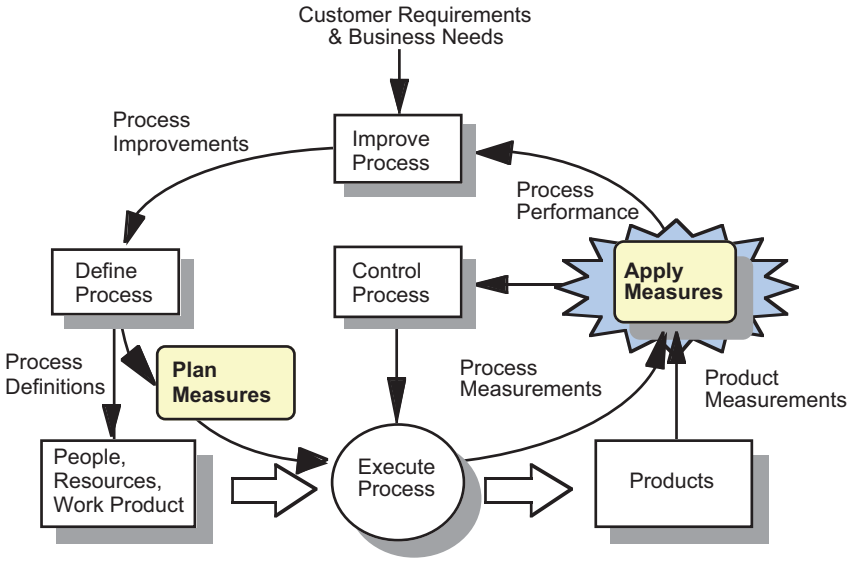


Figure 2.2: An illustration of the relations between software measurements and key responsibilities in project management from SEI Guidebook [25]. Note, that product and process measurements are the only input into the analyses and the process control blocks.

a methodology based on implementation of a continuous software product and process measurement program, that allows for continuous assessment of the software processes variability enabling a "real-time" software process control. Figure 2.2 illustrates their approach.

Hackystat, the "parent" system of STA, is another relevant study that extends the applicability of continuous measurements and confirms the possibility of software process understanding through the analysis of recurrent behaviors [74]. As pointed out by the authors, the visual comprehension of measurements variability and pattern collocations enables "*emergent knowledge that one state variable appears to co-vary with another in the current project context*", allowing for process improvement activities [74].

As an example indicating the possibility of recurrent behaviors discovery through measurements, consider the study by Hindle et al. [66] discussed in the Section 2.5.2 of this chapter that proposes a methodology for recurrent behaviors detection based on Fourier Transform analysis.

STA extends previous approaches built for software measurements analysis by providing an automation for characteristic patterns discovery from software process and product measurements, which, as I expect, shall aid in understanding of recurrent behaviors and their role and effect in

software processes.

2.2 Mining Software Repositories

As mentioned before, mining software repositories is a well established research direction since mid-1970's, when Meir Lehman pioneered the software evolution theory by studying historical records from software repositories [112]. For the last decade, researchers working in the field discuss their approaches and findings in a number of venues. Among these are the Predictive Model in Software Engineering (PROMISE) workshop and the Working Conference on Mining Software Repositories (MSR) which are held within the annual International Conference on Software Engineering (ICSE) and specifically focus on the analysis of software repository artifacts. In order to enable the comparison of proposed techniques performance, both venues encourage researchers to apply them to reference datasets. While PROMISE maintains the same reference dataset over years [65], MSR offers a so-called MSR challenge dataset annually [113] [114]. Note however, that the PROMISE research is mainly concerned with the development of predictive models for Software Engineering [115], whereas MSR traditionally uses data from public software repositories stimulating the diversification of research directions [99] [109] [64].

2.3 Understanding Public Software Repositories

Traditionally, software repositories contain a variety of artifacts produced during the software lifecycle and can be categorized by their purpose. Previous research assigns software repositories into three main categories: source code control, defect tracking, and archived communications systems [108], but other types of repositories exist. These may contain various information, such as software system runtime logs, system testing logs, historical measurements, documentation, tutorials, etc. Recently, a novel type of repositories was proposed for MSR studies – a historical information collected within the community-based question answering service Stack Overflow [114].

As pointed out in previous review studies [99] [64] [116] there are a number of issues associated with mining of *public* repositories which not only create technical difficulties for scientific research,

but also affect its validity. The chief problem is that public project repositories are highly heterogeneous - each is managed and operated mostly in isolation serving a particular project and community needs, therefore having no explicit interactions with other projects. Furthermore, within a project's repository, its SCM subsystems such as version control, defect-tracking, and mailing list, are rarely "connected" [117]. This issue of heterogeneity directly affects MSR studies generality since tools working for and results obtained from one repository, are rarely applicable to another. Yet another issue is that while the public availability of software artifacts mitigates observability and privacy issues, the nature of these artifacts creates a number of other challenges which limit the possible scope of the scientific research and significantly elevate its complexity. Among others, four issues are usually cited as the most significant:

- First of all, the artifacts are created by developers and users not in order to enable scientific research, but rather to support software development activities. Therefore, the informational content of these artifacts is rather poor and additional evidence (i.e., metadata) is often needed [62] [118] [119].

- Second, the majority of these artifacts (change records, defect reports, assigned tasks, etc.) typically represent a snapshot of the software project state rather than reflect any of the performed actions. Thus, it might be simply impossible to infer complete software development processes [120]. Also, this fact effectively renders unusable (within public MSR domain) most if not all of previously developed event-based process and behavior discovery tools as their starting point is an event log [40].

- Third, the project's contributors not only create and submit artifacts to repositories on their own volition, but most of software change management systems (such as Git, Subversion, and Gerrit) encourage an asynchronous workflow where the locally created artifacts may remain uncommitted and therefore unaccounted for, as it has been shown previously [121] [122]. For the same reason, it is often impossible to know *exactly when* the artifact's content was created.

- Finally, the vast volume of produced artifacts, their high dimensionality, and significant noise demand for automated, high throughput and robust analysis techniques [99] [64] [108] [123].

These issues not only create significant external threats to MSR research validity, but usually are *impossible to resolve* without altering the normal flow of OSS software process, for example by implementing a special measurement program, or by introducing instrumented source code editors and development tools (as in Hackystat). Typically, MSR researchers deal with them by seeking for additional evidence in order to support their conclusions [59] [60].

2.3.1 Public software artifacts

Public software repositories offer a wide range of software process and product artifacts for analyses. Among others, these include source code change records, defect reports, feature requests, accepted, rejected and assigned tasks, developer communications, documentation, tutorials, etc. All these allow developers and users to instantly obtain a "snapshot" of the project, i.e., to retrieve the latest (or any previous) source code revision and a complete overview of the software project state, along with the lists of open and closed issues, past and future plans, and other information.

However, while being exceptionally convenient for the project participants, users, and management, this snapshot-oriented nature of public software artifacts creates numerous difficulties for software process research as a "snapshot" rarely reflects finished, ongoing, or planned processes – the issue that limits the feasibility and compromises the validity of performed studies as I have mentioned above.

I acknowledge this software process observability problem when working with public software process artifacts and intentionally avoid discussing and concluding on software processes. Instead, what I shall focus on in this dissertation is the validation of the proposed technique's ability to capture process-characteristic recurrent behaviors when snapshots are viewed in their dynamics.

Nevertheless, I hypothesize that in addition to the fact that the evolution of software measurements in time reflects recurrent development behaviors, some of these can be characteristic of certain aspects of software processes. Therefore, by discovering recurrent patterns in the evolution of software measurements it shall be possible to at least partially infer and evaluate performed software development actions or processes.

Further in this section I review a number of common public software repositories and their arti-

facts to whose measurements STA already has been or potentially can be applied.

2.3.1.1 Source code management system

Source code management systems keep track of the main output of a software project – its source code, which is also the main subject of scientific research. Metrics derived from the source code artifacts are predominant in studies concerned with software evolution, complexity, maintainability, and quality, as well as those that are concerned with productivity, project planning, and cost estimation (i.e., management) [108].

Typically, the evolution of source code is recorded as a sequence of consecutive change records, which are simple artifacts tracking the change of each source code line. Despite the artifact's simplicity, tracing source code evolution through the analysis of change records can become increasingly difficult as developers branch the source code tree, merge it back, or abandon branches [124].

While a large number of metrics can be derived through source code and change records analyses, it offers probably the most functional one – the count of physical lines of code (LOC). Other source code metrics, such as the count of logical lines of code (LLOC), function points (FP), or software system complexity are much less used as they are language-dependent and their derivation involves significant data processing overhead.

2.3.1.2 Defect tracking system

Normally, the software project defect repository serves as a centralized system for managing all of software project Quality Assurance (QA) activities providing users and developers with a means to report and to discuss improper system behavior. In some projects, the defect repository is also used to keep a track of requests for future system features and related discussions.

Artifacts from defect repositories are numerous and complex, as they may contain system logs, input and output files, screen-shots, etc. Their main purpose is to provide users with up to date information about system defects, their severity, and, if implemented in the system, with additional information about their technical nature and resolution plans.

By studying defect records, researchers can address many research questions which are concerned

with software quality, developer's expertise, and the project's technical debt [99]. In addition, defect records are traditionally used for predictive modeling. For example, the ability to build predictive model for future bugs by their association with source code file change patterns (i.e., activity) has been shown by Zimmermann et al. in [103], whether Livshits & Zimmermann have shown a defect predictive model based on characteristic code fragments [125]. An interesting approach for software testing processes optimization based on the identification of source code "hot spots" through mining of the bug reports history has been shown by Ostrand & Weyuker in [126].

2.3.1.3 Developer communications

As OSS projects are usually developed by distributed teams that typically lack the ability for face-to-face meetings, emails, mailing lists, and newsgroups are used as primary communication channels between project participants.

Developer communications artifacts, such as email messages, mailing list posts, and newsgroup messages include agent identification, timestamps, topics, and other data, that provide information allowing for not only process agents identification, but also understanding of their actions and process coordination activities (i.e., roles).

For example Ying et al. in [127] proposed an interesting research direction of mining developer communications content for understanding of software quality, while Huang et al. in [128] used developer communications to build a developer interaction network and to partition developers by level of their involvement into the project or by technical expertise.

2.3.1.4 Q&A websites

Frequently, professional software developers, amateur programmers, and computer hobbyists seek answers to various questions using the Internet. Among others resources, the Internet offers community-driven platforms, such as the Stack Overflow (SO) website that explicitly targets programmers and is dedicated to software-, hardware-, and computer system administration-related issues.

While other types of artifacts available, the ones distributed by SO team are probably the most used in the MSR research. These are distributed monthly and contain the historical information

about questions and answers along with their change history including voting data. In addition, the SO team provides rich metadata about their service contributors. The public Stack Overflow dump was selected as the reference dataset in recent 2013 MSR Challenge [114] which collected a number of submissions proposing interesting data analysis approaches.

While many of these are concerned with programming-related questions, such as identifying topics relevant to particular development communities [129], mining additional technical expertise [130] [131], or identifying problematic APIs [132] [133] and documentation [134], some studies address broad phenomena such as collaborative problem solving [135], knowledge sharing [136] [137], and contributor behaviors [138] [139].

2.3.1.5 Metadata

Often, as reported by Begel et al. [119] who conducted a survey at Microsoft, in order to understand performed software processes, quantitative information about source code change is not sufficient. Through the survey, the authors accounted for 31 types of informational needs necessary for understanding and coordinating software processes, among which the need for the software change metadata was clearly articulated. Amid other reasons, it was found that metadata allows developers to learn the rationale behind software change, find responsible people, discover and track dependencies, and to learn about the status of items in progress. The authors concluded that the majority of developer needs were concerned with people, not the code, and that metadata is essential in meeting these requests.

Similarly, Kim et al. [140] proposed a system for software repositories data collection, storage, and a universal data-exchange language and emphasized the importance of metadata for information management. In addition, the authors showed that it is possible to create a public, metadata-centric system for interfacing closed-source software repositories to public open-source repositories.

Based on these and other results, I have designed the STA database storage in an extensible metadata-centric manner. New types of metadata can be defined by the user and associated with existing and newly collected artifact entities and measurements. In turn, within the process of software trajectories definitions, the metadata allows for efficient data partitioning and retrieval.

2.4 Data assimilation

Currently, there is a voluminous amount of research literature that deals with mining of public software repositories [109] which, in fact, extends an even larger body of research work that covers studies based on mining private software repositories and databases [125] [141] [142].

While the majority of published work is concerned with analyses of a repository information for better understanding of software systems evolution [143] [140], understanding and improving software processes [144], and with studying the impact of software tools on the processes and products [145], some effort has been made towards automation of the historical data retrieval, measurements, and its representation. A number of the proposed solutions allows for the real-time interactive repositories exploration implemented by extending repository management tools such as CVS, SVN, etc. with a front-end engine, such as Bonsai [146], or JReflex [147], while others, such as CVSAnalY [148], softChange [149], and TA-RE [140] propose an approach based on the off-line artifacts retrieval, pre-processing, and on-demand analysis.

Similarly to the latter, STA relies on the off-line retrieval, mirroring, and pre-processing of public software artifacts as shown in Figure 2.3. Note, that since STA has been initially designed as a Hackystat extension [93], it does not need any specific parser and is capable of real-time collection of Hackystat data.

2.5 Relevant MSR research on recurrent behaviors discovery

As I have shown above, MSR is a very diverse research field concerned with a variety of problems. But in this section I focus on previous MSR work that is specifically concerned with the application of analytical techniques to sequences of software artifact measurements – the approach that STA builds upon.

2.5.1 Itemset mining

In data mining, frequently occurring items (actions, events) are often used in order to discover implicit knowledge from large datasets. As I have mentioned earlier in Section 1.3, techniques based

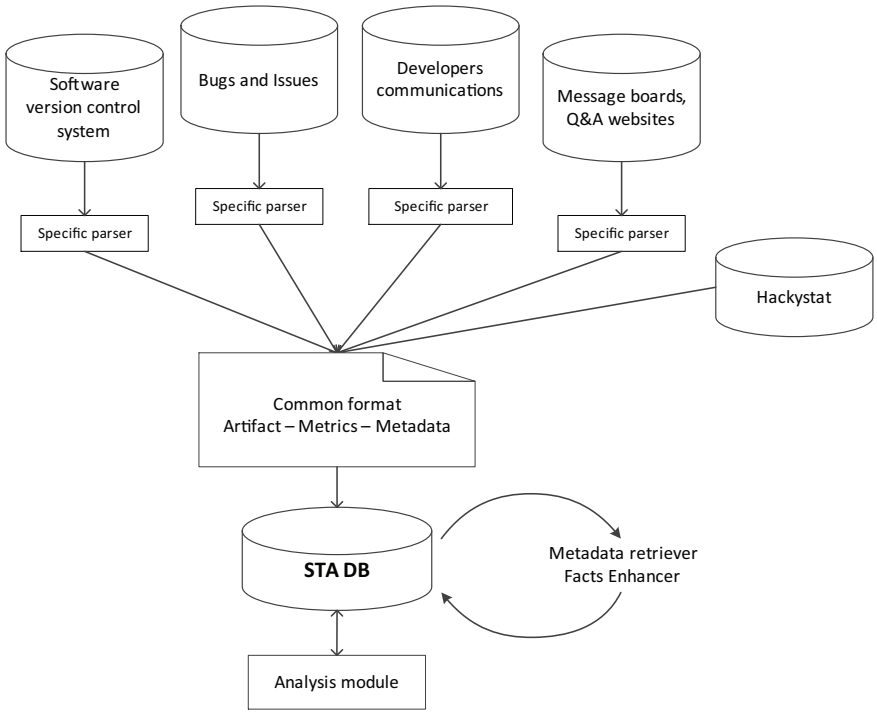


Figure 2.3: A detailed overview of the Software Trajectory Analysis data assimilation layer. At first, software artifacts are mirrored from software repositories, measured, converted into universal to STA format by repository-specific parsers, and stored in the dedicated relational database. In turn, stored in STA DB entities can be further enhanced with additional measurements and metadata.

on frequent items mining were previously applied for software process discovery from development event logs by Cook and Wolf [42] [43] [44] and by Rubin et al. [150]. Unfortunately, since public software repositories do not offer development event logs [120], these techniques can not be adopted for mining software repositories in their proposed form.

Nevertheless, sequential item mining has found a number of applications in MSR. For example Zimmermann et al. in [143] developed a system called ROSE for the identification of co-occurring changes in a software system that aids in future change prediction. For the same purpose, Kagdi et al. [151] developed a sequential pattern mining technique capable of discovery of ordered sequences of frequently changed files. Livshits & Zimmermann [125] developed DynaMine – the system for bug prediction based on mining of frequent function call patterns.

Potentially, itemset mining techniques can be applied to STA results. For example it may be possible to discover ordered, or unordered sequences of recurrent behaviors which can be further

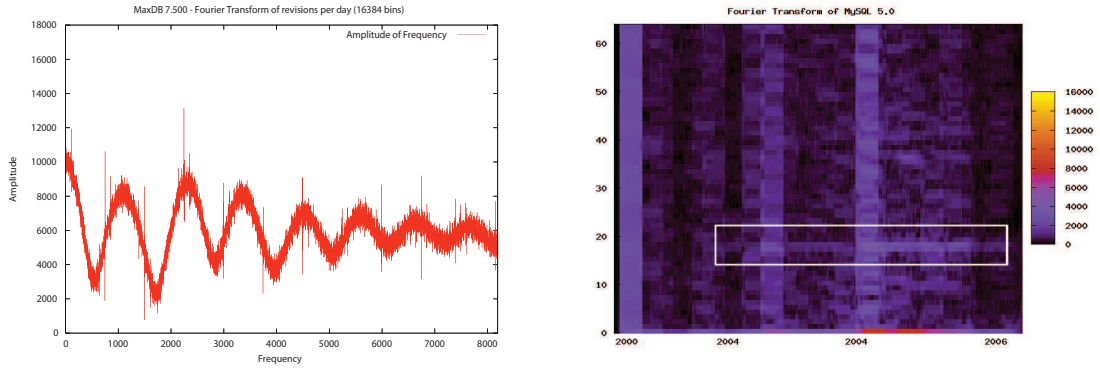


Figure 2.4: Figures from the previous study by Hindle et al. [66] confirming the existence of periodicity in daily changes (left panel) and the possibility of their frequency discovery using Fourier transform (right panel).

associated with particular development actions.

2.5.2 Time series analysis

Because the majority of software artifacts are time stamped, some MSR research seeks to quantitatively analyze ordered in time sequences of software artifacts or their measurements as these may carry useful information about software processes and recurrent behaviors.

For example Herraiz et al. [152] applied Autoregressive Integrated Moving Average (ARIMA) model to software evolution measurements for prediction of future changes. The authors has shown that it is possible to predict a number of future changes in Eclipse by the means of resulting non-explanatory statistical model.

Similarly, Antoniol et al. [69] have explored the application of a common signal processing toolkit built upon Linear Predictive Coding (LPC) and Cepstrum coefficients to modeling of software artifact histories. In particular, the authors have shown that it is possible to identify files with very similar size change histories by using the proposed approach.

The temporal segmentation of time series has been applied to mining of Eclipse change log by Siy et al. [153]. The authors have demonstrated that by partitioning of continuous development activities into the smaller segments whose duration is close to the software release cycle, it is possible to discover "stronger trends" (i.e., characteristic behaviors). For example they have found that developers tend to focus on a particular file subset within a release cycle duration. In addition, they

were able to detect similar change activity patterns among developers.

Finally, Hindle et al. in [66] outlined an approach for discovery of recurrent behaviors from software measurements by Fourier analysis. The left panel of the Figure 2.4 from their work indicates that the studied signal carries potentially distinguishable periodic behaviors, moreover, they were able to detect a promising smear of frequencies between 18 and 19 [days] as it is shown at the right panel. Unfortunately this direction was not further investigated.

2.6 Summary

In this chapter I have reviewed the most relevant to my work previous contributions to the fields of Software Project Management and Mining Software Repositories. Specifically, I have provided the evidence for a tight correlation between product and process evolution and their measurements that enables my research, showed relevant previous work which indicate its feasibility, and enumerated challenges associated with mining of software repositories that shape STA design.

In addition, I have discussed my experiences with earlier STA implementations which confirmed a satisfactory performance of the proposed approach based on measurements partitioning and their symbolic discretization that mitigate for the lack of baselines and noise respectively. Note that at the same time, previous STA experimentation revealed the demand for a new analytical technique that is capable of unsupervised characteristic patterns discovery and ranking. In the next chapter I show a technique called SAX-VSM that addresses the demand and enables unsupervised characteristic patterns discovery from time series.

> Without the right information, you're just another person with an opinion.
>
> Tracy O'Rourke, CEO of Allen-Bradley

CHAPTER 3
INTERPRETABLE TIME SERIES CLASSIFICATION

3.1 Introduction

As I have shown in previous chapters, despite the fact that public software repositories offer a variety of software artifacts and accompanying information for scientific research, their intrinsic complexity and the immaturity of currently available analysis techniques, which often lack generality, automation, and efficiency, limit the breadth and scope of the current MSR research [64] [99].

Addressing this problem, I propose the Software Trajectory Analysis approach – an automated and efficient technique for mining of software repositories, that is specifically concerned with the discovery of recurrent behaviors. This approach is motivated by the evidence that recurrent behaviors are the basic building blocks of software processes [10] [11] [38] and builds upon the hypothesis that it is possible to discover recurrent behaviors by the analysis of a specific data type – "software trajectories" – that are sequences of temporally ordered software artifact measurements (i.e., time series constructed of measurements). While the motivation, background, and evidence leading to this hypothesis were thoroughly discussed in previous chapters, here, I introduce a technique that provides the means for its investigation. For this, I turn to another research field, which is concerned with the analysis of probably the oldest known data type – the time series [154] – and in particular to the research area of Time Series Classification (TSC). Since some of the techniques that have been developed and discussed within TSC research field are concerned with the unsupervised discovery of class-characteristic features, and specifically with the ability to discover *class-characteristic patterns*, which enable the classification, the use of such a technique in STA can be effectively translated into the ability to discover class-characteristic *meaningful* patterns from software trajectories.

Later in this Chapter I shall review the current state of the art in TSC, propose a novel algorithm for *interpretable* time series classification built upon the discovery of class-characteristic patterns, evaluate its performance and the ability to provide an insight into the data and results, and discuss the algorithm's use in STA.

3.2 Time Series classification

Time series classification is a well-established and increasingly popular area of research providing solutions to a wide range of fields, including, but not limited to data mining, image and motion recognition, environmental sciences, health care, and chemometrics. Within the last decade, many time series representations, similarity measures, and classification algorithms have been proposed following the rapid progress in data collection and storage technologies [155]. Nevertheless, to date, the best overall performing classifier in the field is the one nearest-neighbor algorithm (1NN), that can be easily tuned for a particular problem by choosing either a distance measure, an approximation technique, or smoothing [155]. The 1NN classifier is simple, accurate, robust, depends on a few parameters, and requires no training [155] [156] [157].

However, the 1NN technique has a number of significant disadvantages, where the major shortcoming is the inability to offer any insight into the classification results. Another serious limitation is the need for a significantly large training set representing a within-class variance in order to achieve an acceptable accuracy. Finally, while having trivial initialization, the nearest neighbor classification is computationally expensive. Thus, the demand for an *efficient and interpretable* classification technique capable of processing large data volumes remains.

Here, I propose an alternative to 1NN algorithm that addresses the aforementioned limitations. In particular, the proposed technique provides a superior interpretability, learns efficiently from a small training set, and has a low computational complexity.

3.3 Prior and related work in TSC

Almost all of the existing techniques for time series classification can be divided into two major categories [78]. The first category includes techniques based on shape-based similarity metrics where distance is measured directly between time series points. A classic example from this category is the nearest-neighbor classifier built upon Euclidean distance [158] or Dynamic Time Warping (DTW) [159]. The second category consists of classification techniques based on structural similarity metrics which employ a high-level representation of time series, based on their global and/or

local features, for their similarity assessment. Examples from this category include classifiers based on a time series representation obtained with Discrete Fourier Transform [160] or Bag-Of-Patterns [91]. The development of these distinct categories can be explained by the significant difference in their performance: while shape-based similarity techniques are virtually unbeatable on short pre-processed time series [156], they usually fail on data sets that contain long and noisy time series, where structure-based solutions demonstrate the superior performance [91].

Two promising alternatives combining the strengths of techniques from both categories were recently proposed. The first is the Time Series Shapelet approach that allows for a superior interpretability and delivers a compact solution [88]. A shapelet is a short time series "snippet" (i.e., subsequence) that is a representative of class membership and is used for the decision tree construction facilitating class identification and interpretability. In order to find the branching shapelet, the algorithm exhaustively searches for the best discriminatory subsequence on data split via an information gain measure. The algorithm's classification is built upon the similarity measure between the branching shapelet and a full time series, defined as the distance between the shapelet and the closest subsequence in the time series when measured by the normalized Euclidean distance. This exact technique, potentially, combines the superior precision of exact shape-based similarity methods, and the high-throughput classification capacity of feature-based techniques. However, while demonstrating a superior interpretability, robustness, and similar to 1NN algorithm performance, shapelets-based technique is computationally expensive, $O(n^2m^3)$, where n is a number of objects and m is the length of a longest time series, making its adoption for many-class classification problems difficult [161]. While a better solution was recently proposed ($O(nm^2)$), it is an approximate approach based on indexing [162].

The second technique with interpretable results is the nearest neighbor classifier built upon the Bag-Of-Patterns (BOP) representation of time series [91] which is equated to an Information Retrieval (IR) "bag of words" concept and is obtained by extraction, transformation with Symbolic Aggregate approXimation (SAX) [92], and counting the occurrence frequency of short overlapping subsequences (i.e., patterns) along the time series. By applying this procedure to a data set, the algorithm converts it into a vector space, where original time series are represented by the pattern

occurrence frequency vectors. As the authors has shown, these can be classified with a 1NN classifier built with Euclidean distance, or with Cosine similarity that is applied to raw frequencies or their weighted with term frequency-inverse document frequency (i.e., **tf**$*$**idf** [7]) values. BOP classification has several advantages: its complexity is linear ($O(nm)$), it is rotation-invariant since it accounts for local and global structures simultaneously, and it provides an insight into the patterns distribution through frequency histograms. The authors have concluded that the best classification accuracy of BOP-represented time series is achieved by using 1NN classifier based on Euclidean distance.

3.4 SAX-VSM classification algorithm

I propose the time series classification algorithm called SAX-VSM that extends both aforementioned techniques (i.e., shapelet and BOP). In particular, while similar to shapelet-based approaches the algorithm targets the discovery of time series subsequences which are the best characteristic representatives of a class, instead of the iterative search for a class-discriminating shapelet, SAX-VSM ranks by importance all potential candidate subsequences *at once* with a *linear computational complexity* of $O(nm)$. To achieve this, similar to that proposed in BOP, SAX-VSM converts all training time series into bags of SAX words and employs **tf**$*$**idf** for their ranking and Cosine similarity for classification. Nonetheless, instead of building n bags for each of the training time series, SAX-VSM builds a *single bag of words for each of the classes*, which enables effective learning and highly efficient classification ($O(m)$).

As I shall show, these distinct features - the comprehensive summarization of the class' patterns variability with a single bag of words and the ranking of each word class-characterization potential - allow SAX-VSM to achieve a high classification accuracy while providing an exceptional interpretability of the classification results.

3.4.1 Preliminaries

Before describing the algorithm, I shall introduce key terms and concepts used throughout this section, beginning with the data type. Formally speaking, a time series is an ordered sequence

of pairs $T = ((p_1, t_1), (p_2, t_2), ..., (p_i, t_i), ..., (p_m, t_m))$ where values $p_i \in \mathbf{R^n}$ and timestamps are ordered $t_1 < t_2 < ... < t_i < ... < t_m$ and possibly not equidistant, i.e., $|t_i - t_{i-1}| \neq |t_i - t_{i+1}|$.

However, in the research literature, without the loss of generality, equispaced data is typically considered implying that the raw time series can be treated (i.e., interpolated, aggregated, or approximated) in order to become equispaced. Therefore, it is assumed here that the ***time series*** is a vector of scalar observations: $T = (t_1, \ldots, t_m)$, where $t_i \in \mathbf{R^n}$.

Note, that not-equispaced, irregular data is one of the issues when mining software repositories, as I have discussed previously in the Section 2.3, and for this reason STA and SAX-VSM have been designed to effectively mitigate for this: STA aggregates raw measurements into software trajectories first, SAX-VSM aggregates and approximates them second.

In order to rank subsequences by their class-characterization importance, SAX-VSM needs to transform continuous time series data into the symbolic (i.e., discrete) representation at first. The algorithm relies on SAX [4] for discretization and follows the best practices of its application. Specifically, it employs the ***subsequence discretization implemented via a sliding window***, as it is illustrated in Figure 3.1. By sliding a window along the input time series, SAX-VSM extracts short overlapping subsequences and discretizes each of them with SAX. The advantage of this process is that it allows for a better recognition of a localized phenomena as it has been shown in the previous research work targeting motifs (recurrent subsequences) [87] and discords (anomalous subsequences) [163] discovery.

A time series ***subsequence*** of length k of a time series $T = (t_1, t_2, ..., t_m)$ of length m is a time series $T_{i,k} = (t_i, t_{i+1}, ..., t_{i+k-1})$ where $1 \leq i \leq m - k + 1$, i.e., a contiguous fragment of the time series.

Subsequence-based SAX discretization requires three parameters to be provided as the input [4]. Currently, to the best of my knowledge, no efficient solution exists for their optimal selection. In this work I address this problem by using a cross-validation procedure and a parameters optimization scheme based on the dividing rectangles (DIRECT) algorithm that finds optimal parameter values within bounded intervals (i.e., in the range within a minimal and the maximal possible parameter values) [2]. DIRECT is a derivative-free optimization process that possesses local and global op-

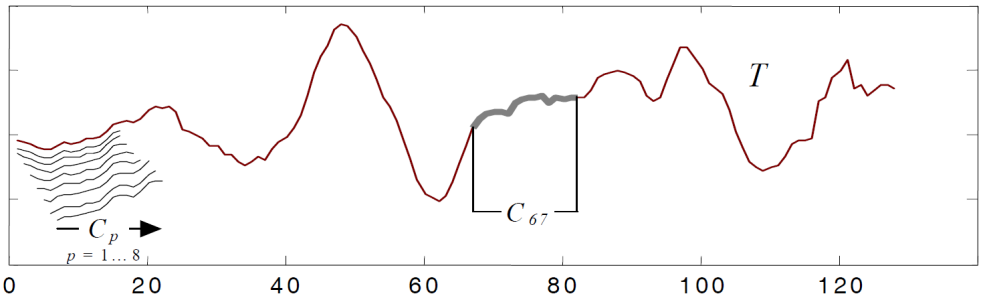


Figure 3.1: An illustration of the sliding window technique from [4]: a time series T of length 128, the subsequence C_{67} (of length p=16), and the first 8 overlapping subsequences extracted by a sliding window.

timization properties; converges relatively quickly, and yields a deterministic, optimized solution. While other optimization techniques exist and some of them may perform better, the performance evaluation of the parameters selection scheme is beyond the scope of my current work.

In the following subsections, I shall review all the techniques which are embedded in SAX-VSM. Subsection 3.4.2 reviews SAX - a symbolic discretization technique, Subsection 3.4.4 discusses numerosity reduction strategies, Subsection 3.4.3 reviews bag of words abstraction. Terms weighting and Vector Space Model are discussed in the Subsection 3.4.5. SAX-VSM algorithm is presented in the Subsection 3.4.6.

3.4.2 Symbolic Aggregate approXimation (SAX)

Discretization of a continuous data into the small number of finite values is highly desirable and often vital for enabling application of machine learning algorithms to datasets reflecting real life phenomena. Hence, probably hundreds of discretization techniques have been developed and are currently available for researchers dealing with knowledge discovery [164]. Among them, the symbolic representation of time series has attracted much attention by enabling the application of numerous string-processing algorithms, bioinformatics tools, and text mining techniques to continuous data.

One of the most popular algorithms for conversion of time series into symbolic representation is the Symbolic Aggregate approXimation [92]. This technique provides a significant reduction of the time series dimensionality and a lower-bounding to Euclidean distance metric, which guarantees no

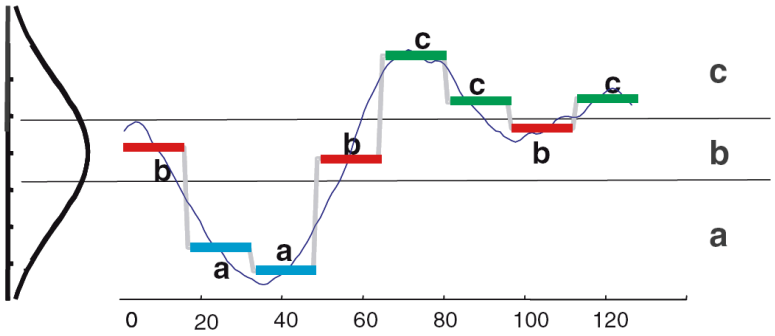


Figure 3.2: An illustration of the SAX approach taken from [4] depicts two pre-determined breakpoints for the three-symbols alphabet and the conversion of the time series of length $n = 128$ into PAA representation followed by mapping of the PAA coefficients into SAX symbols with $w = 8$ and $a = 3$ resulting in the string "**baabccbc**".

false dismissal [4]. These properties are often leveraged by many time series analysis techniques which exploit SAX in order to increase their efficiency. For example, the adoption of SAX indexing allowed for a significantly faster shapelet discovery in [162], although rendering the algorithm approximate.

Given a time series T of a length n, SAX produces its symbolic approximation $\hat{S}$ of a length w where letters are taken from an alphabet A. Along with T, two parameters must be specified as the input: the alphabet size α and the size of the word to produce w. The algorithm, whose overview is shown in Figure 3.2, works as follows.

At first, since it is meaningless to compare time series with different offsets and amplitudes [156], the input time series T is normalized to unit of standard deviation. This normalization procedure, also known as *z-normalization* or "normalization to Zero Mean and Unit of Energy", allows to minimize the effect of the time series amplitude while preserving time series structural specificities [165]. In order to obtain the normalized time series $\widetilde{T}$, the input time series mean is subtracted from each point and the resulting value is divided by their standard deviation:

$$\widetilde{t_i} = \frac{t_i - \mu}{\sigma}, i \in 1, .., n \tag{3.1}$$

If, however, the standard deviation value falls below a fixed threshold, the normalization procedure is not applied in order to avoid a possible over-amplification of the background noise, as it has been

Table 3.1: An example of the SAX alphabet lookup table that contains the breakpoints dividing a Gaussian distribution in an arbitrary number (from 2 to 11) of equiprobable regions.

β_i \ α	2	3	4	5	6	7	8	9	10	11
β_1	0,00	-0,43	-0,67	-0,84	-0,97	-1,07	-1,15	-1,22	-1,28	-1,34
β_2		0,43	0,00	-0,25	-0,43	-0,57	-0,67	-0,76	-0,84	-0,91
β_3			0,67	0,25	0,00	-0,18	-0,32	-0,43	-0,52	-0,60
β_4				0,84	0,43	0,18	0,00	-0,14	-0,25	-0,35
β_5					0,97	0,57	0,32	0,14	0,00	-0,11
β_6						1,07	0,67	0,43	0,25	0,11
β_7							1,15	0,76	0,52	0,35
β_8								1,22	0,84	0,60
β_9									1,28	0,91
β_{10}										1,34

shown in [4].

At the second step, the dimensionality of the normalized time series is reduced to w by obtaining its Piecewise Aggregate Approximation (PAA). For this, $\widetilde{T}$ is transformed into a vector of PAA coefficients C ($|C| = \omega$) by dividing it into equal-sized segments and computing their mean values:

$$c_i = \frac{w}{n} \sum_{j=\frac{n}{w}(i-1)+1}^{\frac{n}{w}i} \widetilde{t}_j \tag{3.2}$$

Note, that for any L_p norm this transformation satisfies to a lower-bounding condition and guarantees no false dismissals [3] [166].

Discretization is performed at the final step of the SAX algorithm where each of the PAA coefficients obtained at the previous step is converted into a letter $\widehat{c}$ of the alphabet A by the use of lookup tables (as shown in Table 3.1) which define a list of breakpoints $B = \beta_1, \beta_2, ..., \beta_{a-1}$ such that $\beta_{i-1} < \beta_i$ and $\beta_0 = -\infty$, $\beta_a = \infty$ that divide the area under $N(0,1)$ into a equal areas. The design of these tables rests on the assumption that normalized time series tend to have Gaussian distribution [167] [92]. By assigning a corresponding alphabet symbol α_j to each interval $[\beta_{j-1}, \beta_j)$, the conversion of the vector of PAA coefficients C into the string $\widehat{C}$ implemented as follows:

$$\widehat{c}_i = \alpha_j, \text{ if } c_i \in [\beta_{j-1}, \beta_j) \tag{3.3}$$

Table 3.2: An example of the MINDIST function lookup table for the $a = 11$

	a	b	c	d	e	f	g	h	i	j	k
a	0,00	0,00	0,43	0,73	0,99	1,22	1,45	1,68	1,94	2,24	2,67
b	0,00	0,00	0,00	0,30	0,56	0,79	1,02	1,26	1,51	1,82	2,24
c	0,43	0,00	0,00	0,00	0,26	0,49	0,72	0,95	1,21	1,51	1,94
d	0,73	0,30	0,00	0,00	0,00	0,23	0,46	0,70	0,95	1,26	1,68
e	0,99	0,56	0,26	0,00	0,00	0,00	0,23	0,46	0,72	1,02	1,45
f	1,22	0,79	0,49	0,23	0,00	0,00	0,00	0,23	0,49	0,79	1,22
g	1,45	1,02	0,72	0,46	0,23	0,00	0,00	0,00	0,26	0,56	0,99
h	1,68	1,26	0,95	0,70	0,46	0,23	0,00	0,00	0,00	0,30	0,73
i	1,94	1,51	1,21	0,95	0,72	0,49	0,26	0,00	0,00	0,00	0,43
j	2,24	1,82	1,51	1,26	1,02	0,79	0,56	0,30	0,00	0,00	0,00
k	2,67	2,24	1,94	1,68	1,45	1,22	0,99	0,73	0,43	0,00	0,00

SAX also introduces a new metric for measuring the distance between strings by extending the Euclidean and PAA [3] distances. The function returning the minimal distance between two symbolic representations of the original time series $\widehat{Q}$ and $\widehat{C}$ is defined as

$$\text{MINDIST}(\widehat{Q}, \widehat{C}) \equiv \sqrt{\frac{n}{w}} \sqrt{\sum_{i=1}^{w} (dist(\widehat{q}_i, \widehat{c}_i))^2} \tag{3.4}$$

where the $dist$ function is implemented by using lookup tables specific to a set of used breakpoints (alphabet size) as shown in Table 3.2, and where the singular value for each cell (r, c) is computed as

$$cell_{(r,c)} = \begin{cases} 0, \text{ if } |r - c| \leq 1 \\ \beta_{\max(r,c)-1} - \beta_{\min(r,c)-1}, \text{ otherwise} \end{cases} \tag{3.5}$$

As shown by Lin et al. [92], the SAX distance metric is lower-bounding to the PAA distance, i.e.

$$\sum_{i=1}^{n} (q_i - c_i)^2 \geq n(\bar{Q} - \bar{C})^2 \geq n(dist(\hat{Q}, \hat{C}))^2 \tag{3.6}$$

The SAX lower bound was later examined by Ding et al. [168] and was found to be superior in precision to the spectral decomposition methods on non-periodic data sets while only "slightly" inferior to other techniques on periodic data. This findings and the capacity of SAX to be tuned for data specificities made it the best option for symbolic discretization step of SAX-VSM.

3.4.3 Bag of words representation of time series

Following its introduction, SAX was shown to be an efficient tool for solving problems of finding time series motifs (recurrent patterns) and discords (anomalous patterns) in time series [87, 163]. The authors employed a sliding window-based subsequence extraction technique and augmented data structures (hash table in [87] and trie in [163]) in order to index observed SAX words. Further, by analyzing their occurrence frequencies and locations, they were able to capture frequent and rare SAX words representing motifs and discords subsequences respectively. Later, the same technique based on the combination of sliding window and SAX was used in the numerous works, most notably in time series classification using bag of patterns (BOP) [91] and in the Fast-Shapelet algorithm [162].

I also use this sliding window technique to convert a time series T of a length n into the set of m SAX words, where $m = (n - l_s) + 1$ and l_s is the sliding window length. By sliding a window of length l_s across time series T, extracting subsequences, converting them into SAX words, and placing these words into an unordered collection, the algorithm builds the *bag of words* representation of the original time series T.

3.4.4 SAX numerosity reduction

Previously, the analysis of SAX-based algorithms performance by Keogh et al. [87] and Lin et al. [163] revealed that the best matches for a sliding window subsequence tend to be its neighbors, specifically the subsequence one point to the right and the subsequence one point to the left – due to the smoothing effects of PAA approximation and SAX discretization. The authors defined these matching subsequences as *trivial matches* and found that in a smooth region of a time series the amount of trivial matches can be large enough to dominate over true matches due to the over-counting – an issue which may significantly bias the result and even make it meaningless [169] for SAX-based techniques. Hence, they have concluded, when extracting subsequences from the time series via a sliding window the trivial matches should be excluded.

The authors proposed a sampling strategy based on a *MINDIST* (Eq. 3.4) distance function designed in order to avoid the trivial and degenerate solutions. If l consecutive SAX words

$\widehat{S}_{i,k}, \widehat{S}_{i+1,k}, ..., \widehat{S}_{i+l-1,k}$ corresponding to subsequences $T_{i,k}, T_{i+1,k}, ..., T_{i+l-1,k}$ extracted with sliding window have been found equal when using *MINDIST*, they kept only the first entry $\widehat{S}_{i,k}$. The authors also noted that, similarly to the run length encoding data compression technique, if one would ever need to retrieve all the occurrences of $\widehat{S}_{i,k}$, they can be found by sliding the window from the first occurrence to the right until the word which is different from $\widehat{S}_{i,k}$ is found.

While the authors found the inclusion of the numerosity reduction vital for motif and discord discovery applications, intuitively, since SAX-VSM deals with the classification, an aggressive numerosity reduction may in fact reduce the classification performance as it has been shown in the original BOP work [91]. Moreover, by the design of **tf**$*$**idf** statistics (Eq. 3.10), the over-counting effect is significantly mediated by the inverse document frequency **idf** that efficiently reduces the effect of high word counts proportionally to their inter-class occurrence.

In order to clarify this issue, I have conducted an exploratory study of the SAX numerosity reduction effect on SAX-VSM performance. In a series of experiments, I have found, that for most of used data sets, the application of numerosity reduction significantly reduces the DIRECT scheme convergence time and, sometimes, improves the classification accuracy. Furthermore, once I have relaxed the trivial match constraints by the substitution of *MINDIST* with a distance function based on the Hamming distance [170], I was able to slightly improve the classification accuracy for more than half of the data sets used for SAX-VSM performance evaluation as shown in Table 3.6 (SAX-VSM accuracy results obtained with the "exact" value of numerosity reduction parameter). The *HAMMING* distance function for two SAX words $\hat{Q}$ and $\hat{C}$ of the same length w is defined as the count of letters in which they differ:

$$\begin{aligned} &\text{HAMMING}(\widehat{Q}, \widehat{C}) \equiv \sum_{i=1}^{w} I(\widehat{q}_i, \widehat{c}_i), \\ &\text{where } I(\widehat{q}_i, \widehat{c}_i) = \begin{cases} 1, \text{ if } \widehat{q}_i \neq \widehat{c}_i \\ 0, \text{ if } \widehat{q}_i = \widehat{c}_i \end{cases} \end{aligned} \tag{3.7}$$

For further use, I abbreviate the numerosity reduction strategy based on the previous work (i.e., on *MINDIST* function) as ***CLASSIC***, while the one based on *HAMMING* distance as ***EXACT***.

Note that, as the experimental evaluation has shown, the effect of the numerosity reduction strategy may or may not be significant for a particular dataset, moreover, since this effect is impossible to know in advance, the numerosity reduction strategy becomes yet another parameter which needs to be properly selected in order to achieve the best SAX-VSM performance for a given dataset. Therefore, in total, there are four parameters which need to be optimized for the SAX-VSM application to a particular dataset.

3.4.5 Vector Space Model (VSM) adaptation

I use the Vector Space Model exactly as it is known in Information Retrieval (IR) [171] for manipulations with abstracted by SAX words time series subsequences.

Similarly to IR, I define and use the following expressions:

- *term* - a single SAX word;
- *bag of words* - an unordered collection of SAX words, i.e., terms;
- *corpus* - a set of bags;
- *term frequency matrix* - a matrix defining the term occurrence frequency for each bag, whose rows correspond to all observed in a corpus terms and whose columns correspond to bags;
- *term weight matrix* - a similar to term frequency matrix structure defining the weight coefficient of a term for each of the corpus' bags;
- document (bag) *term weight vector* - a column of the weight matrix defining weights of all terms for a single bag.

Note however, that I use terms *bag of words* and *document* for abbreviation of an unordered collection of SAX words interchangeably, while in IR these usually bear different meaning as a *document* presumes words ordering (i.e., semantics). Although similar definitions, such as *bag of features* [172] or *bag of patterns* [91], were recently proposed for techniques built upon SAX [91], I use the traditional *bag of words* definition since it reflects my workflow best.

Table 3.3: The SMART notation.

Term frequency	**Document frequency**	**Normalization**
n (natural): $\text{tf}_{t,d}$	**n** (no): 1	**n** (none): 1
l (logarithm): $1 + log(\text{tf}_{t,d})$	**t** (idf): $log\frac{N}{df_t}$	**c** (cosine): $\frac{1}{\sqrt{w_1^2+w_2^2+\ldots+w_M^2}}$
a (augmented): $0.5 + \frac{0.5\times\text{tf}_{t,d}}{\max(\text{tf}_{t,d})}$	**p** (prob idf): $\textbf{max}\left(0, \log\frac{N-df_t}{df_t}\right)$	**b** (byte size): $\frac{1}{CharLength^{\alpha}}, \alpha < 1$
b (boolean): $\begin{cases}1, & \text{if } \text{tf}_{t,d} > 0\\ 0, & \text{otherwise}\end{cases}$		
L (log average): $\frac{1+\log(\text{tf}_{t,d})}{1+\log(\text{ave}_{t\epsilon d}(\text{tf}_{t,d}))}$		

Given a training set of time series, that is typically built from labeled sets of time series (i.e., "classes"), SAX-VSM builds a single bag of SAX words for each of the classes by processing all class' time series with a sliding window and SAX. Then, these bags are combined into a corpus which in turn is transformed into the term frequency matrix, whose rows correspond to the set of all SAX words (terms) found in *all classes*, whereas each column denotes a class of the training set. Each element of this matrix is an observed frequency of a term in a class. Note, that because SAX words extracted from time series of one class are often not among other classes, as it is shown further in the Section 3.7.2, this matrix is usually sparse.

Following to the common in IR workflow, SAX-VSM employs the **tf**$*$**idf** weighting scheme [173] for each element of this matrix in order to transform the frequency value into a weight coefficient. The **tf**$*$**idf** weight for a term is defined as a product of two factors: term frequency (**tf**) and inverse document frequency (**idf**). For the first factor, I use logarithmically scaled term frequency (Table 3.3) [173]:

$$\textbf{tf}_{t,d} = \begin{cases} 1 + \log(\mathbf{f}_{t,d}), & \text{if } \mathbf{f}_{t,d} > 0 \\ 0, & \text{otherwise} \end{cases} \tag{3.8}$$

where t is a term and d is a bag of words (the document in IR terms), and $\mathbf{f}_{t,d}$ is a frequency of the term in the bag. For the second factor I use inverse document frequency:

$$\textbf{idf}_{t,D} = \log_{10}\frac{|D|}{|d \in D : t \in d|} = \log_{10}\frac{N}{\textbf{df}_t} \tag{3.9}$$

where N is the cardinality of corpus D (the total number of bags) and the denominator $\mathbf{df}_t$ is a number of bags where the term t appears.

Thus, the **tf∗idf** value for a term *t* in the document *d* of a corpus *D* is defined as:

$$\mathbf{tf} * \mathbf{idf}(t, d, D) = \mathbf{tf}_{t,d} \times \mathbf{idf}_{t,D} = \log(1 + \mathbf{f}_{t,d}) \times \log_{10} \frac{N}{\mathbf{df}_t} \tag{3.10}$$

for all cases where $\mathbf{f}_{t,d} > 0$ and $\mathbf{df}_t > 0$, or zero otherwise. Once all terms of a corpus are weighted, the term frequency matrix becomes a term weight matrix and its columns are used as the class' *term weight vectors* that facilitate the classification with Cosine similarity.

The Cosine similarity measure between two vectors is defined by their inner product and magnitude. For two vectors $\mathbf{a}$ and $\mathbf{b}$ that is:

$$\text{similarity}(\mathbf{a}, \mathbf{b}) = cos(\theta) = \frac{\mathbf{a} \cdot \mathbf{b}}{||a|| \cdot ||b||} = \frac{\sum_{i=1}^{n} a_i \times b_i}{\sqrt{\sum_{i=1}^{n} (a_i)^2} \times \sqrt{\sum_{i=1}^{n} (b_i)^2}} \tag{3.11}$$

3.4.6 SAX-VSM implementation

As many other time series structure-based classification techniques, SAX-VSM consists of two phases - the training (i.e., learning of class-characteristic patterns) and the classification. Within the training phase, SAX-VSM discretizes all labeled time series with SAX and builds N bags of SAX words, where N is the number of classes. Then, by applying the **tf∗idf** weighting scheme to the corpus of N bags it obtains N weight vectors which it uses for the time series classification procedure built upon the Cosine similarity. The schematic overview of the algorithm training and classification phases is given in Figure 3.3.

3.4.6.1 Training

SAX-VSM training starts by the transformation of all labeled time series into SAX representation. This process is configured by four parameters: the sliding window length (*W*), the number of PAA segments per window (*P*), SAX alphabet size (*A*), and the numerosity reduction strategy. Note that each subsequence extracted with a sliding window is normalized (Sec. 3.4.2) before being processed

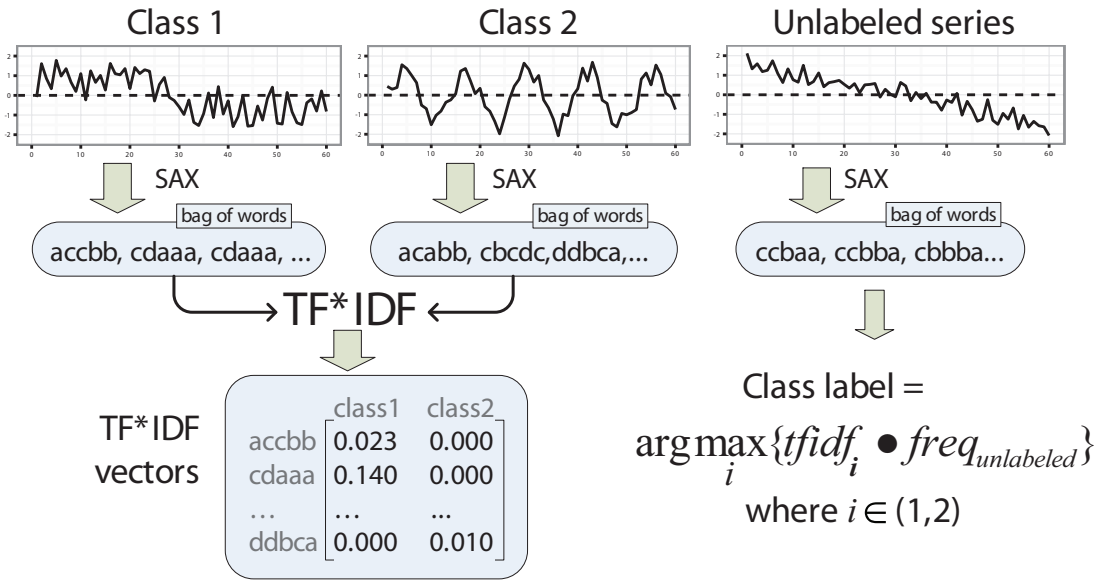


Figure 3.3: An overview of the SAX-VSM algorithm: at first, labeled time series are converted into bags of words using SAX; secondly, **tf** $*$ **idf** statistics is computed resulting in a single weight vector per training class. For the classification, an unlabeled time series is converted into the term frequency vector and assigned a label of the weight vector that yields a maximal cosine similarity value. This is ***ltc.nnn*** weighting schema in SMART notation (Table 3.3).

with PAA, however, if the standard deviation value falls below a fixed threshold, the normalization is not applied in order to avoid over-amplification of the background noise [92].

By applying this procedure to all time series from N training classes, the algorithm builds a corpus of N word bags. Then, it computes weights of all of the corpus' terms using **tf**$*$**idf** and outputs N real-valued weight vectors of equal length representing training classes.

Because the whole training set must be processed, training of SAX-VSM classifier is computationally expensive ($O(nm)$, where n is the number of time series and m is the maximal length of the time series). However, there is no need to maintain an index of training time series, or to keep any of them in the memory at runtime – the algorithm simply iterates over all training time series building bags of SAX words incrementally. Once built and weighted with **tf**$*$**idf** , the corpus is discarded – only the resulting set of N real-valued weight vectors is retained for the classification.

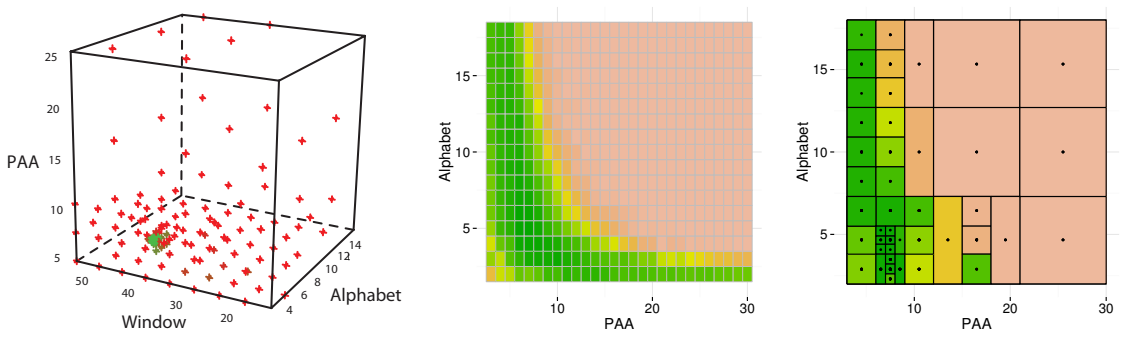


Figure 3.4: An illustration of the DIRECT-driven SAX-VSM parameters optimization for *SyntheticControl* dataset. The left panel shows all points sampled by DIRECT in the space $PAA * Window * Alphabet$. The red points correspond to high error values while green points correspond to low error values in cross-validation experiments. Note the green points concentration at W=42. Middle panel shows the classification error heat map obtained by a complete scan of all 432 points of the hypercube slice when W=42. Right panel shows the classification error heat map of the same slice when the parameters search was optimized by DIRECT, the optimal solution (P=8,A=4) was found by sampling just 43 points.

3.4.6.2 Classification

In order to classify an unlabeled input time series, SAX-VSM transforms it into a terms frequency vector using exactly the same sliding window technique and SAX parameters set that were used for the training. Then, it computes cosine similarity values between this vector and N **tf**$*$**idf** weight vectors that represent training classes. The input time series is assigned to the class whose vector yields the maximal cosine similarity value.

3.5 Parameters optimization

As shown above, in total, SAX-VSM requires four discretization parameters to be specified upfront from which three (the sliding window length, the PAA size, and the SAX alphabet size) may vary in a wide range. Unfortunately, to the best of my knowledge, there is no efficient solution known for their selection.

Addressing this issue I propose a solution based on a common cross-validation and DIRECT (DIviding RECTangles) optimization scheme [1]. As I shall show, the combination of these techniques allows for an optimal parameter selection while using only the training data. For brevity, I

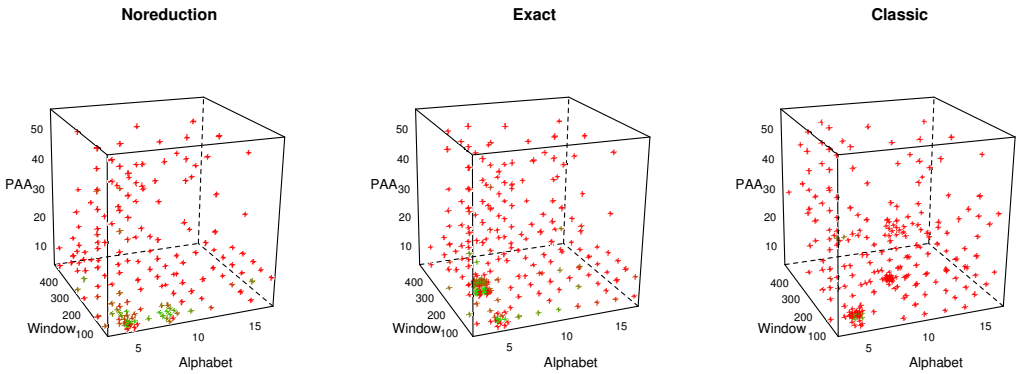


Figure 3.5: An illustration of the numerosity reduction strategy effect on DIRECT-driven parameters optimization process. The points represent the error rate in cross-validation experiments and are colored according to its value: the red color indicates high error values, while the green corresponds to low error values. Note that dense points collocations are differ among strategies, which indicates the difference in their error function gradient.

omit the detailed explanation of the DIRECT algorithm background and motivation, referring the interested reader to the original work [2] for additional details.

DIRECT is designed to deal with a parameters optimization problems of form:

$$\min_{x} \; f(x), \; f \in \mathbf{R}, \; x, X_L, X_U \in \mathbf{R}, \; \text{where } X_L \leq x \leq X_U \tag{3.12}$$

where $f(x)$ is the error function, and x is the parameters vector. At the first step DIRECT scales the search domain to the unit hypercube. The function is then evaluated at the center point of the hypercube. As pointed in [2], computing the function value at the center is an advantage of the method when dealing with problems in higher dimensions. Then, DIRECT iteratively performs two procedures - partitioning the hypercube into smaller hyper-rectangles and identifying a set of potentially-optimal ones by sampling their centers. At each step, the function is evaluated at the center points of all potentially-optimal hyper-rectangles. The procedure continues interactively until the error function converges. Note, that DIRECT is guaranteed to converge to the global optimal function value, as the number of iterations approaches to infinity [2].

Since DIRECT is designed to search for global minima of a real valued function over a bound constrained domain, whereas SAX parameters are natural numbers, I employ the rounding of a reported solution values to the nearest integer. Figure 3.4 illustrates the application of leave-one-out cross-validation and DIRECT to the *SyntheticControl* data set [174] which consists of 6 classes. In this case, the algorithm converged after sampling just 130 out of 13,860 possible parameters combinations – that is over 100x speedup.

Figure 3.5 shows the effect of each of the numerosity reduction strategies on parameters optimization process with DIRECT for Beef dataset that features time series classes obtained by measuring a degree of beef contamination by adulterants with mid-infrared spectroscopy [175]. Sixteen iterations of DIRECT were performed for each experiment. The optimal solution (Window=19, PAA=17, Alphabet=3) was found with *EXACT* numerosity reduction strategy in 8 iterations; without numerosity reduction, optimization process converged in 10 iterations, while with *MINDIST*-based numerosity reduction in 9 iterations. Note the differences in sampled locations between strategies: without numerosity reduction, DIRECT efficiently found the minima location after sampling of 317 locations, whether with reduction, a number of close to optimal locations was found earlier and thus sampled more rigorously. 365 locations were sampled with *MINDIST*-based numerosity reduction, and 369 locations with Hamming-based numerosity reduction, which indicates an increase in the optimization process sensitivity when a numerosity reduction is used.

Note, that the parameters optimization scheme discussed above does not include the numerosity reduction strategy. The reasons for this is that the numerosity reduction strategy mostly affects the parameters optimization scheme convergence speed rather than the accuracy of the classification, thus, for the most cases, it can be simply pre-defined to *EXACT*.

3.6 Intuition behind SAX-VSM

First, by combining ***all*** SAX words extracted from ***all*** time series of single class into a ***single bag*** of words, SAX-VSM manages to effectively capture and summarize the observed intraclass variability even from a small training set.

Second, by normalizing, smoothing, approximating time series subsequences, and discarding

Table 3.4: Description of the datasets used in performance evaluation.

Class type Dataset	Datasets
Image data	50 words, Adiac, Yoga , Face Four, Face all, Faces UCR, Fish, Swedish Leaf, OSU Leaf, Arrow Head, Shield, Diatom, Medical Images
Motion data	Gun-Point, Cricket, Cricket-NEW, Sony AIBO walk, Pass Graph, uWaveGesture
Spectroscopy data	Beef, Coffee, Olive Oil, Wheat
Synthetic datasets	Cylinder-Bell-Funnel, Synthetic Control, Two Patterns, Mallat
Energy consumption	Italy Power Demand, Electrical Devices
Medical measurements	ECG200, ECG 5 days, Medical images, ECG Thorax
Other measurements obtained with instruments	Trace, Lightning 2, Lightning 7, Wafer, Ford A, Ford B, Chlorine concentration, Starlight

their original ordering, SAX-VSM focuses exclusively on the local structural phenomena regardless of the data distortion by the rotation and its corruption by the noise or values loss.

Third, **tf∗idf** statistics naturally highlights terms that are unique to the class by assigning them high weights, whereas terms that observed in multiple classes are assigned low, inversely proportional to their interclass presence, weights. This improves the selectivity of the classification by decreasing the contribution of "confusive" multi-class terms, while increasing the contribution of unique "class-defining" terms to the final similarity measurement value.

Ultimately, the algorithm compares the set of subsequences extracted from an unlabeled time series with the weighted set of all characteristic subsequences representing the whole of the training class. Thus, an unknown time series is classified by its similarity not to a given number of "neighbors" as in kNN or BOP classifiers, or to a single characteristic subsequence, as in shapelet-based classifier, but by the *combined similarity* of all its subsequences to all known discriminative patterns found in the whole of the class.

3.7 SAX-VSM performance evaluation

I have proposed a novel algorithm for time series classification based on SAX approximation of time series and Vector Space Model called SAX-VSM. In this section I describe a set of experiments assessing its performance and exploring its ability to provide an insight into the classification results.

Table 3.5: Classification accuracy comparison for state of the art nearest-neighbor, interpretable, and SAX-VSM classifiers.

Dataset	Num. of classes	1NN-Euclidean	1NN-DTW	Fast Shapelets	Bag Of Patterns	SAX-VSM
Adiac	37	0.389	0.391	0.514	0.432	**0.381**
Beef	5	0.467	0.467	0.447	0.433	**0.3**
CBF	3	0.148	0.003	0.053	0.013	**0.002**
Coffee	2	0.250	0.180	0.067	0.036	**0.0**
ECG200	2	**0.120**	0.230	0.227	0.140	0.140
FaceAll	14	0.286	**0.192**	0.402	0.219	0.207
FaceFour	4	0.216	0.170	0.089	0.011	**0.0**
Fish	7	0.217	0.167	0.197	0.074	**0.017**
Gun-Point	2	0.087	0.093	0.060	0.027	**0.007**
Lightning2	2	0.246	**0.131**	0.295	0.164	0.196
Lightning7	7	0.425	**0.274**	0.403	0.466	0.301
Olive Oil	4	**0.133**	**0.133**	0.213	**0.133**	**0.133**
OSU Leaf	6	0.483	0.409	0.359	0.236	**0.107**
Syn.Control	6	0.120	**0.007**	0.081	0.037	0.010
Swed.Leaf	15	0.213	0.210	0.270	**0.198**	0.251
Trace	4	0.240	**0.0**	0.002	**0.0**	**0.0**
Two patterns	4	0.090	**0.0**	0.113	0.129	0.006
Wafer	2	0.005	0.020	0.004	0.003	**0.0006**
Yoga	2	0.170	**0.164**	0.249	0.170	**0.164**

3.7.1 Analysis of the classification accuracy

I have evaluated SAX-VSM accuracy on 45 datasets, whose majority was taken from the benchmark data disseminated through the UCR repository [174]. These datasets represent a variety of data types that reflect typical TSC domain problems. Table 3.4 describes their origin.

Table 3.5 compares the classification accuracy of SAX-VSM with previously published results for four competing classifiers: two state-of-the-art 1NN classifiers based on Euclidean distance and DTW, and two interpretable classifiers based on recently proposed Fast-Shapelets technique [162] and BOP [91] on 19 datasets. I have selected these particular techniques in order to position SAX-VSM in terms of the classification accuracy and the results interpretability.

Table 3.6 compares the classification accuracy of SAX-VSM with 1NN state of the art classifiers based on Euclidean and DTW distances on all 45 datasets. Fast-Shapelet and BOP classifiers were excluded from this comparison table because their performance for these datasets is unknown.

Table 3.6: Classification accuracy comparison for state of the art nearest-neighbor and SAX-VSM classifiers.

Dataset	Num. of classes	Training set size	Testing set size	Series length	1NN-Euclidean	1NN-DTW	SAX-VSM	Discretization param.
Synthetic Control	6	300	300	60	0.12	**0.007**	0.0133	45,7,5,exact
CBF	3	30	900	128	0.148	0.003	**0.0021**	55,4,12,nored
Gun Point	2	50	150	150	0.087	0.093	**0.066**	32,12,9,exact
50 words	50	450	455	270	0.369	**0.310**	0.3582	190,10,3,exact
Trace	4	100	100	275	0.24	**0.0**	**0.0000**	220,16,11,exact
Adiac	37	390	391	176	0.389	0.396	**0.3810**	100,24,16,nored
Yoga	2	300	3000	426	0.170	**0.164**	**0.1639**	70,14,15,nored
Beef	5	30	30	470	0.467	0.5	**0.3**	19,17,3,exact
Coffee	2	28	28	286	0.25	0.179	**0.0**	107,22,3,nored
Olive Oil	4	30	30	570	**0.133**	**0.133**	**0.1330**	460,52,13,classic
ECG200	2	100	100	96	**0.12**	0.23	0.1400	44,9,5,exact
ECG 5 days	2	23	861	136	0.065	0.232	**0.0100**	41,11,4,exact
Face all	14	560	1,69	131	0.286	**0.192**	0.2065	42,8,4,nored
Face four	4	24	88	350	0.216	0.170	**0.1112**	67,7,5,exact
Fish	7	175	175	463	0.217	0.167	**0.0171**	99,19,8,nored
Swedish Leaf	15	500	625	128	0.213	**0.210**	0.2512	49,9,7,exact
OSU Leaf	6	200	242	427	0.483	0.409	**0.0867**	33,8,12,nored
Lightning 2	2	60	61	637	0.246	0.131	**0.1967**	169,15,3,nored
Lightning 7	7	70	73	319	0.425	0.274	**0.3287**	97,17,3,nored
Wafer	2	1	6,174	152	0.005	0.020	**0.0010**	34,32,7,classic
Two Patterns	4	1	4	128	0.09	**0.0**	0.0040	107,12,3,nored
Ford A	2	3,601	1,32	500	0.3182	0.484	**0.1272**	80,10,5,exact
Ford B	2	3,636	810	500	0.4086	0.495062	**0.2567**	80,10,5,exact
Chlorine Concentration	3	467	3840	166	0.35	0.352	**0.3341**	30,27,5,classic
Cricket	2	9	98	166	0.0511	**0.0102**	**0.0102**	165,10,4,exact
Cricket - NEW	2	9	98	166	0.4375	**0.125**	0.2343	165,10,4,exact
Sony AIBO walk	2	20	601	70	0.3045	0.2745	**0.2628**	54,4,16,exact
PassGraph	2	69	131	364	0.3664	0.2824	**0.2812**	119,10,15,nored
Wheat Spectrography	7	49	726	1050	0.44	0.457	**0.2790**	130,50,10,nored
Arrowhead	3	36	175	625	0.32	0.32	**0.3028**	113,11,3,classic
Shield	3	30	129	1179	0.1395	0.1395	**0.0772**	150,12,4,nored
Mallat	8	320	2080	256	**0.0235**	0.0312	0.0274	214,10,15,nored
uWaveGesture_X	8	896	3582	315	**0.2607**	0.2725	0.2635	260,7,5,exact
uWaveGesture_Y	8	896	3582	315	**0.3384**	0.3659	0.3534	240,10,4,exact
uWaveGesture_Z	8	896	3582	315	0.3504	0.3417	**0.3400**	258,8,4,exact
Diatom Size Reduction	4	16	306	345	0.0654	**0.0327**	0.0653	174,15,18,exact
Medical Images	10	381	760	99	0.3158	**0.2631**	0.4802	29,9,5,exact
Words Synonyms	25	267	638	270	0.3824	**0.3511**	0.4404	198,10,3,exact
FacesUCR	14	200	2050	131	0.2307	0.0951	**0.0751**	38,8,3,exact
Symbols	6	25	995	398	0.1005	**0.0503**	0.1015	112,12,5,exact
Starlight Curves	3	1000	8236	1024	**0.0632**	0.093	0.0807	172,15,11,exact
Italy Power Demand	2	67	1029	24	0.0949	**0.0495**	0.1166	13,16,5,exact
ElectricalDevices	7	8953	7745	96	0.9132	0.9132	**0.3227**	17,13,6,nored
ECG Thorax1	42	1800	1965	750	0.171	**0.209**	0.2340	44,15,14,exact
ECG Thorax2	42	1800	1965	750	0.120	**0.135**	0.1450	44,15,14,exact

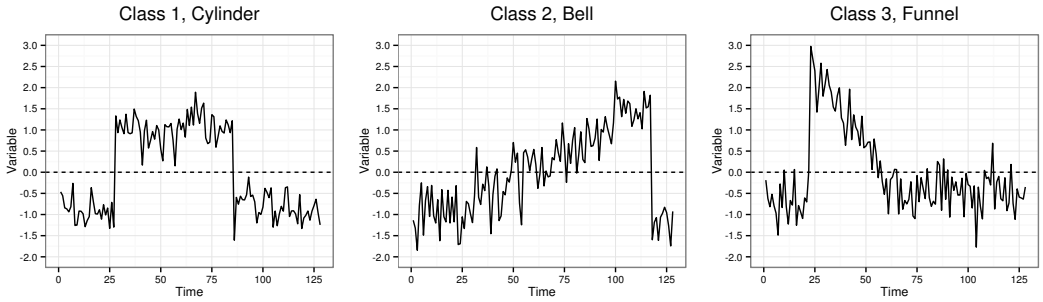


Figure 3.6: An example of the three classes from CBF dataset.

Note, that in the evaluation, I followed the train/test data split as provided by UCR. At first, the train data was used in the cross-validation for optimization of SAX parameters using DIRECT. Second, once found, the optimal parameter settings were used to assess SAX-VSM classification accuracy on the test data. The last column of table Tables 3.5 and 3.6 reports the SAX-VSM classification accuracy and the parameter settings.

3.7.2 Scalability analysis

For synthetic data sets, it is possible to create as many instances as one needs for the experimentation. Moreover, the ground truth corresponding to their features and patterns is always known through their design. I have used Cylinder-Bell-Funnel [176] and Two Patterns [177] synthetic datasets in order to investigate and to compare the performance of SAX-VSM and 1NN Euclidean classifier on increasingly large data sets.

3.7.2.1 Cylinder-Bell-Funnel (CBF) dataset

The CBF problem was introduced in [176] and since then has been routinely used in TSC for the investigation of a classifier performance behavior. The dataset represents a classical problem of time series classification where the class assignment is made upon the detection of a *single global* pattern. The goal is to separate three classes of objects: cylinder (c), bell (b), and funnel (f). Figure 3.6 shows examples of time series from each of the classes. The Cylinder is characterized by a plateau, the Bell by an increasing linear ramp followed by a sharp drop, while the Funnel is characterized by a sharp rise followed by a gradual decrease. The class-characteristic feature start,

its duration (length of the plateau and ramps) and the amplitude are randomized. Gaussian noise is also added to each time series point:

$$
\begin{aligned}
c(t) &= (6+\eta)\cdot\chi_{[a,b]}(t)+\epsilon(t) \\
b(t) &= (6+\eta)\cdot\chi_{[a,b]}(t)\cdot(t-a)/(b-a)+\epsilon(t) \\
f(t) &= (6+\eta)\cdot\chi_{[a,b]}(t)\cdot(b-t)/(b-a)+\epsilon(t)
\end{aligned}
\quad \text{, where } \chi_{[a,b]} = \begin{cases} 0, t < a \\ 1, a \leq t \leq b \\ 0, t > b \end{cases} \tag{3.13}
$$

where η and $\epsilon(t)$ are drawn from a standard normal distribution $N(0,1)$, a is an integer drawn uniformly from the interval $[16,32]$ and $(b-a)$ is drawn uniformly from $[32,96]$.

3.7.2.2 Two patterns dataset

As mentioned, the CBF problem demands a classifier to make the decision based on a single global pattern. Contrary, the Two Patterns problem requires a classifier to recognize ordered occurrences of two local patterns.

In particular, patterns that are used to define classes are the upward step and the downward step, as it is shown in Figure 3.7. Class DD corresponds to two downward steps, DU to the succession of a downward and an upward step, etc. The position and the duration of these patterns are randomized, which creates an additional challenge for a classifier to distinguish classes with similar patterns, i.e., UD and DU. The signal surrounding patterns is randomized with the Gaussian noise. As pointed by the dataset author, this problem is particularly challenging for classical learning algorithms that do not account for the sequential measurements dependency [177].

3.7.3 Classification scalability

In a series of experiments, I varied the training data set size from 5 to $1,600$ instances of each time series class, while the test data set size remained fixed to $10,000$ instances. For small training sets, SAX-VSM was found to be significantly more accurate than 1NN classifier based on Euclidean distance but less accurate than 1NN classifier based on DTW. However, by the time there were more than 400 time series in a training set, there was no statistically significant difference in accuracy between all classifiers, as shown at left panels of Figures 3.8 and 3.9.

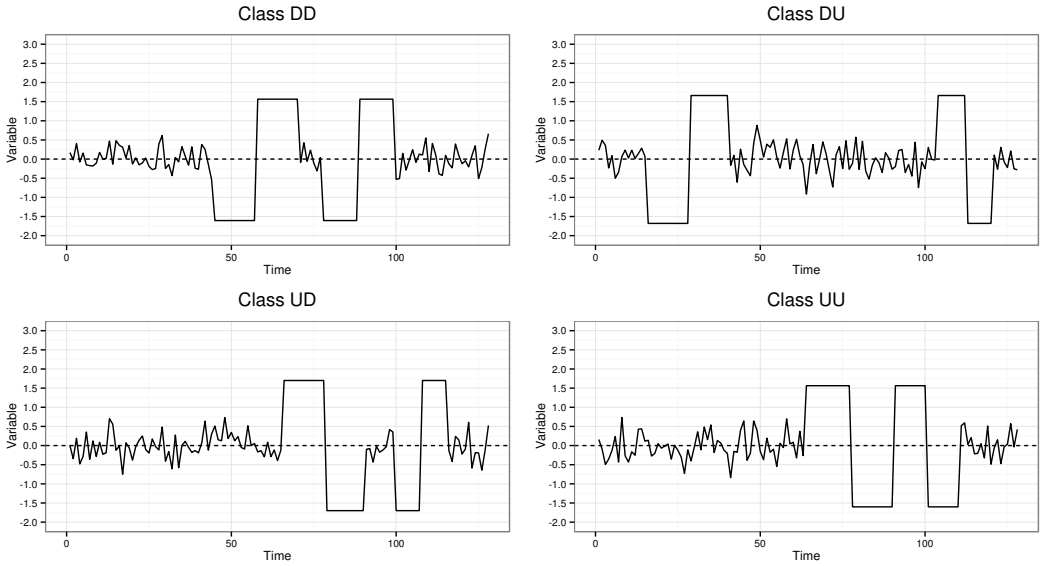


Figure 3.7: An example of the four classes from Two Patterns dataset.

As per the running time cost, to no surprise, the DTW-based classifier was found to be the most expensive technique. Due to the comprehensive training, SAX-VSM was found to be more expensive than 1NN Euclidean classifier on small training sets, but outperformed it on larger training sets.

However, SAX-VSM can perform the training offline and can load class-characteristic **tf∗idf** weight vectors when needed. If this option can be utilized, the proposed classifier performs significantly faster than both 1NN classifiers as shown at the right panels of Figures 3.8 and 3.9.

3.7.3.1 SAX-VSM training scalability

In another series of experiments I have investigated the scalability of the algorithm with unrealistic training set sizes - up to one million of instances of each of CBF classes. As expected, with the growth of the training set size, the curve for a total number of distinct SAX words and curves for dictionary sizes of each of CBF classes reflected a significant saturation as it is shown at the left panel of Figure 3.10. For the largest of training sets - 10^6 instances of each class - the size of the dictionary peaked at $67,324$ of distinct words (which is less than 10% of all possible words of length 7 from an alphabet of 7 letters), and the largest **tf∗idf** vector accounted for $23,569$ values (Figure

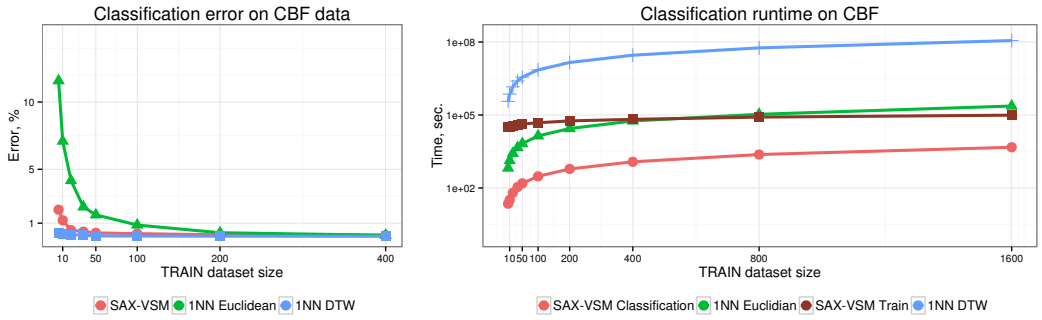


Figure 3.8: Classification accuracy and run time comparison for SAX-VSM and 1NN classifiers on CBF data. SAX-VSM performs significantly better than 1NN Euclidean classifier with a limited amount of training samples, but not as good as 1NN DTW classifier (left panel). While SAX-VSM is fastest in the classification, its performance is comparable to 1NN Euclidean classifier when the training time is accounted for (right panel).

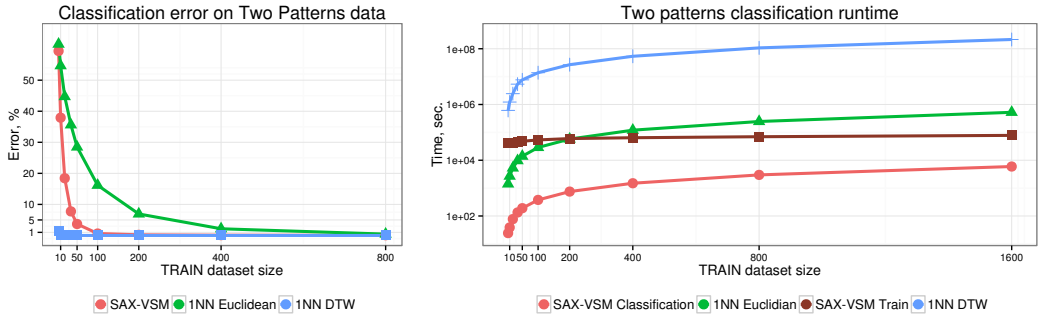


Figure 3.9: Classification accuracy and run time comparison for SAX-VSM and 1NN classifiers on Two Patterns data. Experiment reveals that on small training set sizes the problem is much harder for SAX-VSM and 1NN Euclidean classifiers than it is for the 1NN DTW classifier. Nevertheless, similarly to the previous experiment, SAX-VSM performs better than 1NN Euclidean classifier in terms of the both: accuracy and speed.

3.10, right). In my opinion, this result reflects two characteristics of the data set chosen: the first is that the diversity of words which are possible to encounter in CBF dataset is quite limited by its classes configuration (i.e., single global pattern) and by the choice of SAX parameters (smoothing). The second specificity is that IDF (Inverse Document Frequency, 3.9) efficiently limits the growth of dictionaries by eliminating those words, which are observed in all classes.

The similar behavior was observed in the experimentation with Two Patterns dataset. The Figure 3.11 shows the rapid saturation of SAX word dictionaries as a training dataset grows in size.

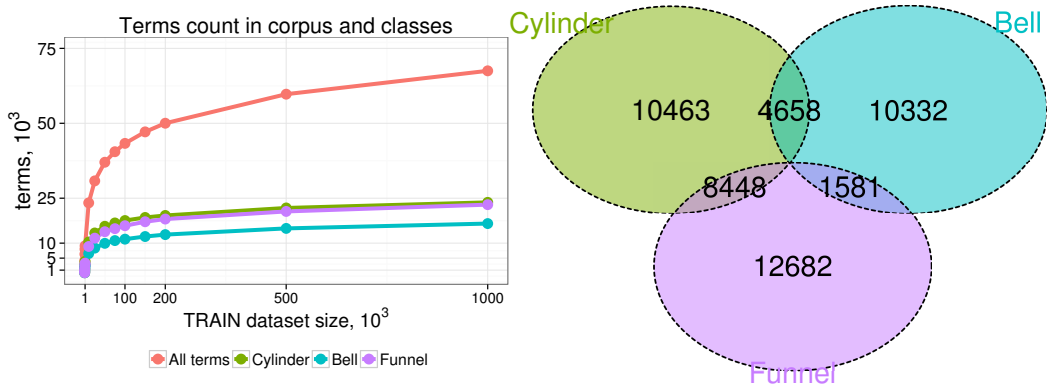


Figure 3.10: An illustration of the SAX-VSM class-characteristic pattern vectors size evolution for the CBF dataset with increasingly large training set size (left panel), and the distribution of terms in the CBF corpus for a training set of one million time series of each class.

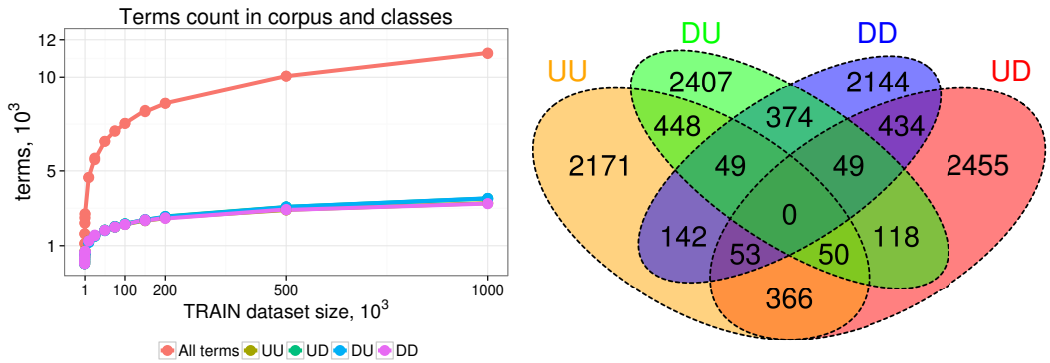


Figure 3.11: An illustration of the SAX-VSM class-characteristic pattern vectors size evolution for the Two Patterns dataset with increasingly large training set size (left panel), and the distribution of terms in the Two Patterns corpus for a training set of one million time series of each class.

3.7.4 Robustness to noise

As shown, the growth of the dimensionality of **tf∗idf** weight vectors follows the growth of the training set size, which indicates that SAX-VSM is continuously learning from the observed class variability. Since the weight of each of overlapping subsequences extracted from time series via sliding window contributes only a small fraction to the final similarity value, and since each subsequence represents a localized structural phenomenon, intuitively, the SAX-VSM classifier shall be robust to the noise and to the partial signal loss. In this case, the cosine similarity between two high dimensional weight vectors may not degrade significantly enough to cause misclassification (Equation 3.11).

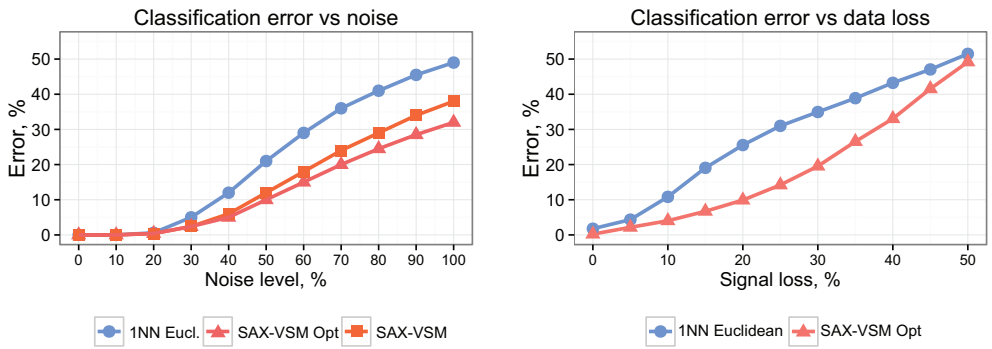


Figure 3.12: An illustration of the SAX-VSM classification performance evolution on CBF dataset with added noise (left panel, the random noise amplitude varies up to 100% of that of the signal value), and with a signal loss (right panel, the start and stop of the"lost interval" were chosen randomly). *SAX-VSM Opt* curves correspond to the results obtained with the "optimized" for each case SAX parameters.

In one series of experiments, by fixing a training set size to 250 time series, I have varied the standard deviation of Gaussian noise in CBF model (whose default value is about 17% of a signal level). I have found, that SAX-VSM increasingly outperformed 1NN Euclidean classifier with the growth of the noise level (Fig.3.12 Left). Further improvement of SAX-VSM performance was achieved by the tuning of the PAA smoothing through a gradual increase of the sliding window size proportionally to the growth of the noise level (Fig.3.12 Left, *SAX-VSM Opt* curve).

In another series of experiments, I replaced up to 50% of an unlabeled time series span with a randomly placed stretches of the Gaussian noise, mimicking the signal corruption. Again, SAX-VSM performed consistently better than 1NN Euclidean classifier regardless of the training set size, which I have varied from 5 to 1'000. The *SAX-VSM Opt* curve at Fig.3.12 (Right) depicts an experiment where the training set size was fixed to 50 time series of each class and when the sliding window size was decreased inversely proportionally to the signal loss growth.

3.7.5 Interpretable classification

While the classification performance results in previous sections confirms that SAX-VSM classifier has a comparable to state of the art classification performance, its major strength is in the level of allowed interpretability of classification results.

Previously, in the original shapelets work [88, 89], it has been shown that the resulting decision

tree offers an insight into the data specificity through class-characteristic patterns. In the successive work based on shapelets [161], it was also shown that the discovery of multiple shapelets provides increasingly better resolution and intuition into the interpretability of classification.

However, as the authors noted, the runtime cost of multiple shapelets discovery in a many class problems can be prohibitive to the approach applicability. In contrast, SAX-VSM extracts and weights all patterns at once, without any added cost. Therefore, it could be the only choice for interpretable classification in many class problems.

Further in this section, I propose a SAX-VSM based heatmap-like time series class specificity visualization that provides insight into the classification result and show the utility of the subsequence ranking for interpreting of the class-characteristic data specificity.

3.7.5.1 Heatmap-like visualization

Since SAX-VSM builds **tf∗idf** weight vectors using all subsequences extracted from a training set, it is possible to find out the weight of any arbitrary selected subsequence. This feature enables a novel visualization technique that can be used to gain an immediate insight into the layout of "important" class-characterizing subsequences as it is shown in Figures 3.13 and 3.14.

In order to highlight class-characteristic subsequences, the color hue value for each point is computed as the combination of **tf∗idf** weights of all subsequences that span the point. If the subsequence is found to be characteristic to other than the analyzed time series class, its weight is subtracted, if it belongs to the same class, the weight is added.

This type of visual analysis allows for an immediate insight into the classification results as for any of the classified time series it is possible to visualize which subsequences were found class-characteristic for each of the classes and to which degree.

3.7.5.2 Gun Point data set

By following the previously mentioned shapelet-based work [88] [161], I have used a well-studied *Gun/Point* data set [178] to explore the interpretability of classification results. This data set contains two classes: time series in the *Gun* class corresponds to the actor's hand motion when drawing a

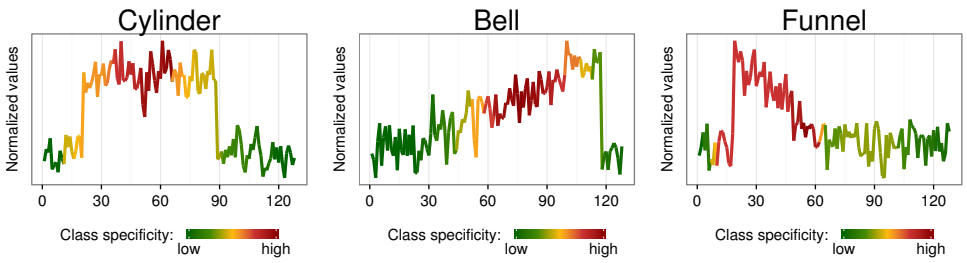


Figure 3.13: An example of the heatmap-like visualization that exploits SAX-VSM subsequence ranking in order to highlight time series segments that are highly characteristic to the class. Highlighted by the visualization features corresponding to a sudden rise, plateau, and a sudden drop in Cylinder, increasing trend in Bell, and to a sudden rise followed by a gradual drop in Funnel, align exactly with the design of these classes [176].

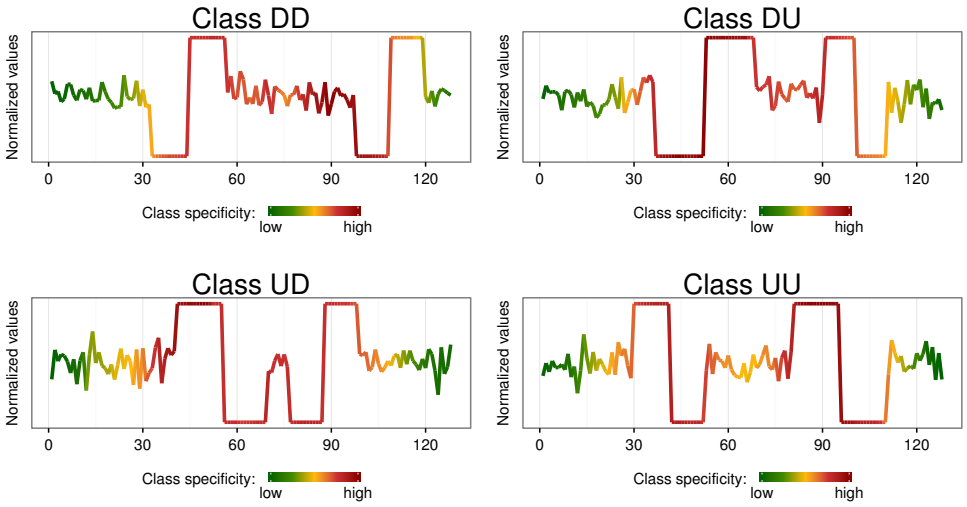


Figure 3.14: An example of the heatmap-like visualization for Two Patterns dataset, which also confirms the proposed algorithm's ability to capture the class specificity in more challenging than CBF settings where class-characteristic patterns are local and ordered [177].

replicate gun from a hip-mounted holster, pointing it at the target for a second, and returning the gun to the holster; time series in the *Point* class corresponds to the actor's hand motion when pretending of drawing a gun — the actor points her index finger to a target for about a second, and then returns the hand to her side.

Similarly to previously reported results [88] [161], SAX-VSM captured all distinguishing features as shown in Figure 3.15. The most weighted by SAX-VSM pattern in *Gun* class corresponds to fine extra movements required to lift and aim the prop. The most weighted pattern in *Point* class

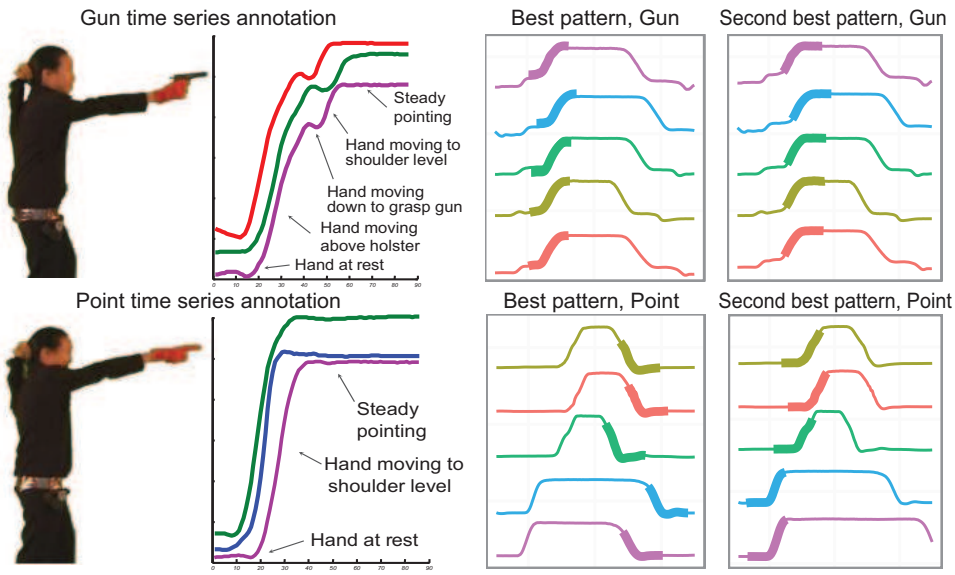


Figure 3.15: Best class-characteristic subsequences (right panels, bold lines) discovered by SAX-VSM in the *Gun/Point* data set. Left panels show actor's stills and the time series annotation made by an expert while the right panels show locations of characteristic subsequences. Note, that while the upward arm motion found to be more "important" in the *Gun* class (gun retrieval and aiming), the downward arm motion better characterizes the *Point* class (note the "overshoot" phenomena in propless arm return). This result aligns with previous work [88] and [161]. (Stills and annotation are used with a permission from E. Keogh)

corresponds to the "overshoot" phenomena that is causing the characteristic dip in the time series. Also, similarly to the original *GunPoint* work [178], as second to the best pattern in *Point* class, SAX-VSM highlighted the lack of distinguishing subtle extra movements required for lifting a hand above the holster and reaching down for the gun.

3.7.5.3 OSU Leaf data set

According to the original data source, A.Grandhi [180], with the growth of digitized data volumes, there is a huge demand for automatic management and retrieval of various images. The *OSULeaf* data set consist of curves obtained by image segmentation and boundary extraction (in the anti-clockwise direction) from digitized leaf images of six classes: *Acer Circinatum, Acer Glabrum, Acer Macrophyllum, Acer Negundo, Quercus Garryana* and *Quercus Kelloggii*. The authors of the original work were able to solve the problem of leaf curve classification by using the nearest neighbor classifier built upon DTW distance achieving 61% of the classification accuracy.

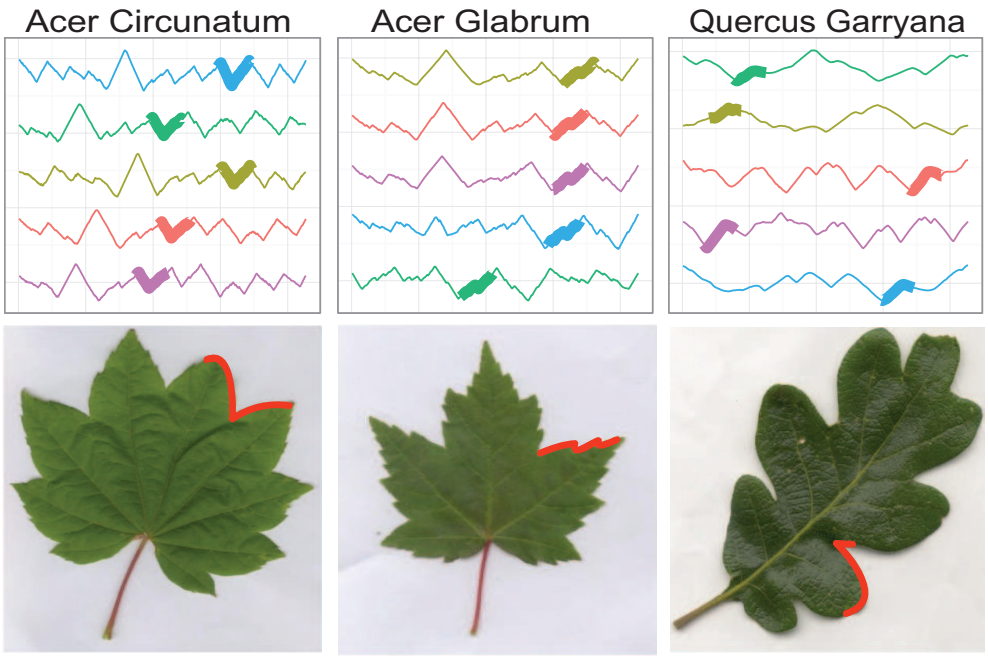


Figure 3.16: The best class-characteristic subsequences (top panels, bold lines) discovered by SAX-VSM in the *OSULeaf dataset*. These patterns align exactly with well known in botany leaves discrimination techniques by the lobe shape, serration, and tip type [179].

Since SAX-VSM performed significantly better on this problem, I have investigated the classification results. In contrast to NN classification results that do not offer any insights, SAX-VSM application yielded a set of class-specific characteristic patterns for each of six classes of leaves from *OSULeaf* data set. Further patterns investigation revealed, that they closely match known techniques for leaves classification based on their shape and margin [179]. Highlighted by SAX-VSM features include the slightly lobed shape and acute tips of Acer Circinatum leaves, the serrated blade of Acer Glabrum leaves, the acuminate tip and a characteristic serration of Acer Macrophyllum leaves, the pinnately compound leaves arrangement of Acer Negundo, the incised leaf margin of Quercus Kelloggii, and the lobed leaf structure of Quercus Garryana. Figure 3.16 shows a subset of these characteristic patterns and the original leaf images with highlighted features that correspond to SAX-VSM discovered patterns.

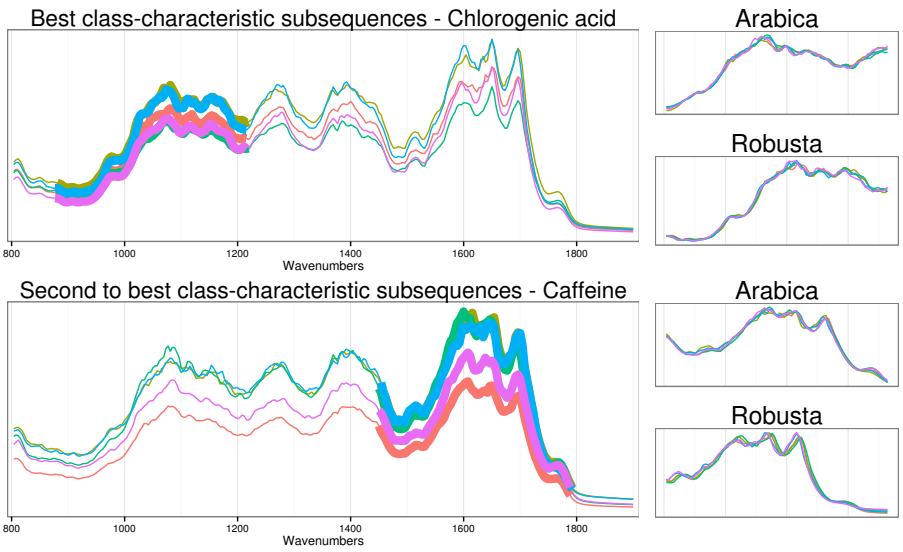


Figure 3.17: The best class-characteristic subsequences (left panels, bold lines) discovered by SAX-VSM in the *Coffee data set*. Right panels show zoom-in view on these subsequences in Arabica and Robusta spectrograms. These discriminative subsequences correspond to the chlorogenic acid (best subsequence) and to the caffeine (second to best) regions of spectra. This result aligns with the ground truth and the original work based on PCA [79] exactly.

3.7.5.4 Coffee data set

Another illustration of interpretable classification with SAX-VSM is based on the Coffee dataset [79]. The time series for this problem were obtained with the Fourier transform infrared spectroscopy instrument equipped with a diffuse reflection sampling station (DRIFT). The raw time series were truncated to 286 data points which represent the observed spectra within the 800-1900 cm^{-1} range.

The two top-ranked by SAX-VSM subsequences in both datasets correspond to spectrogram intervals accounting for abundances of Chlorogenic acid (the best characteristic pattern) and Caffeine (the second to best characteristic pattern). These two chemical compounds are known to be responsible for the flavor differences in Arabica and Robusta coffees; moreover, these spectrogram intervals were also reported as discriminative when used in the PCA-based classification technique developed by the authors of the original work [79].

3.7.5.5 Characteristic pattern utility

As shown above, via discovered by SAX-VSM class-characteristic patterns we can learn the inherent structure of the analyzed data in a manner that allows intuitive interpretation of classification results. In addition, ranked class-characteristic pattern vectors provide a compact way to summarize data classes.

Note, that in contrast to shapelet-based techniques, which are based on the single class-characteristic pattern, SAX-VSM generates a ranked list of patterns, which, once computed, allows much deeper insight into the studied phenomena through the examination of second best, third, and so on, patterns. When compared with BOP approach, where a list of ranked patterns is built for each class' entity, SAX-VSM, which aggregates patterns into a single bag, provides a naturally better way to summarize the class-characteristic specificity.

3.8 Clustering

Clustering is a generic technique used for data partitioning, visualization, and exploration. In addition, clustering is an important subroutine in many data mining algorithms [181]. Since clustering algorithms are built upon a distance function, that computes similarity between clustered entities, the algorithm's performance is highly dependent on the performance of the chosen distance function. Thus, an experimental evaluation of the proposed in this chapter technique in clustering shall provide an additional perspective on its performance and the applicability beyond the classification.

3.8.1 Hierarchical clustering

Probably, one of the most used clustering algorithms is hierarchical clustering which requires no parameters to be specified as input [182]. It computes pairwise distances between all objects and produces a nested hierarchy of clusters offering the efficient data partitioning and visualization.

Previously, it has been shown that the bag-of-patterns time series representation along with the Euclidean distance provide superior clustering performance[91]. For comparison, I have performed a similar experiment that only differ in the time series representation and the distance metric – I

have used **tf∗idf** weight vectors obtained from SAX-VSM and the Cosine similarity. Confirming previous work, I have found, that the combination of SAX and Vector space model outperforms classical shape-based distance metrics. For example, Figure 3.18 depicts the result of a hierarchical clustering of the data subset from *SyntheticControl* dataset. Obviously, the data partitioning obtained with SAX-VSM clustering is superior those based on Euclidean and DTW distance metrics as it properly splits data into three valid branches.

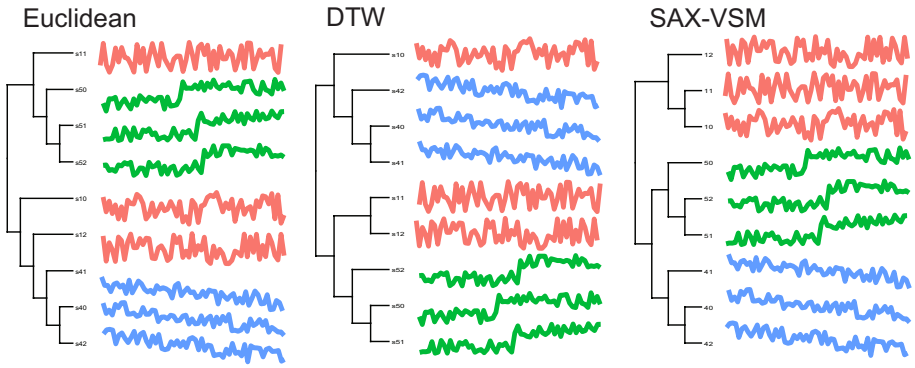


Figure 3.18: A comparison of the distance metrics performance in hierarchical clustering for the subset of three *SyntheticControl* classes: *Normal, Decreasing trend*, and *Upward shift*. The Euclidean distance and Dynamic Time Warping were applied to raw time series while the Cosine similarity was applied to their representation as term weights vectors. Complete linkage was used to generate clusters. Only SAX-VSM was able to partition the data properly.

3.8.2 k-Means clustering

Another popular choice for data partitioning is k-Means clustering algorithm [183]. The basic intuition behind this algorithm is that through the iterative reassignment of objects into different clusters the intra-cluster distance is minimized.

As it has been shown before, k-Means algorithm scales much better than hierarchical partitioning techniques [184]. In addition, k-Means clustering is also well studied in IR field. For example in [185], the authors extensively examined seven different criterion functions for partitional document clustering and found, that *k*-prototypes partitioning with cosine dissimilarity (the approach similar to SAX-VSM) delivers an excellent performance.

Following this work, I have implemented a similar to [186] *spherical k-means algorithm* and

found, that it converges quickly and delivers a satisfactory partitioning on short synthetic data sets. Further, I have evaluated the technique on the long time series from PhysioNet archive [187], from which I have extracted 250 time series corresponding to five vital signals: two ECG leads (aVR and II), RESP, PLETH, and CO2 waves, trimming them to 2'048 points. Similarly to BOP experimentation [91], I have applied a reference k-Means algorithm implementation based on the Euclidean distance [188] [189] to this dataset achieving the maximum clustering quality of 0.39, when measured as proposed in [190] on the best clustering (the one with the smallest objective function in 10 runs). SAX-VSM based spherical k-Means implementation outperformed the reference technique yielding clusters with the quality of 0.67, confirming the superior performance of the combination of weighted subsequence based time series representation and Cosine similarity.

3.9 Conclusions an discussion

In this Chapter, I have proposed a novel interpretable technique for time series classification that is based on class-characteristic patterns discovery. As I have shown above and summarized in Table 3.7, that SAX-VSM is competitive with, or superior to, other classification techniques on a variety of classical data mining problems. In addition, I have described a number of advantages of the proposed algorithm over existing structure-based time series classification techniques emphasizing its capacity to discover and rank short subsequences by their class characterization power.

By an experimental evaluation, I have shown that this particular feature – the ability to discover and rank class-characteristic subsequences – can be exploited for data mining and machine learning purposes. In such contexts, SAX-VSM can be used as an exploratory tool that aids in the discovery of data set characteristic patterns. Therefore, its application for software trajectory characteristic patterns discovery problem is natural. Similar to that in the discussed previously classification problems of CBF, Two Patterns, Coffee, OSU Leaf, and Gun/Point, I expect SAX-VSM to be capable to highlight software trajectory subsequences that can be easily interpreted and attributed to characteristic behaviors associated with particularities of software processes.

Table 3.7: Comparison of time series classification algorithms characteristics.

Classification algorithm	Training required?	Accuracy	Classification efficiency	Major strengths	Major weaknesses
1-NN Euclidean	no	low	slow	fast start	slow classification
1-NN DTW	no	highest	slowest	fast start, the best accuracy	very slow classification and parameters optimization
Fast Shapelets	yes	low	fast	some interpretability, superior compactness, fast classification	very slow training
Bag Of Patterns	yes	high	fast	interpretability, fast classification	unintuitive parameters
SAX-VSM	yes	high	fast	superior interpretability, classifier' compactness, fast classification	slow parameters optimization

The ability to focus attention on important things is a defining characteristic of intelligence.

Robert J. Shiller.

CHAPTER 4
RESULTS

In preceding chapters, I have discussed a number of phenomena which provide the motivation for my exploratory study investigating the possibility of recurrent behaviors discovery from software artifacts, reviewed the relevant previous work from the research field of software repository mining, identifying unexplored and under-explored directions, and proposed a novel generic temporal data-mining technique called SAX-VSM, which, potentially, can automate the discovery of recurrent behaviors from software artifact measurements.

In this chapter, I shall present, evaluate, and discuss SAX-VSM-based implementation of the Software Trajectory Analysis framework (STA) that provides an end-to-end generic and customizable solution for the problem of recurrent behaviors discovery from software trajectories. As I shall show, throughout my exploratory study STA has evolved from a narrow focused tool to a universal framework that facilitates software artifacts collection, their measurements, software trajectories construction, and, the most importantly, enables the recurrent behaviors discovery.

4.1 Software Trajectory Analysis system overview

Before presenting and discussing the current STA implementation, I shall briefly review its background starting with the software trajectory definition.

Recall, that the *software trajectory* is defined as an abstract representation of the software product and/or process evolution by a series of temporally ordered measurements. In other words, it is a field-specific abstraction that technically is the time series with attached contextual meaning. Intuitively, this abstraction in Software Engineering is similar to that used in Physics, where trajectory is an approximate path that a moving object draws in a physical space, or in Mathematics, where trajectory is defined as a reduced in complexity sequence of states of a dynamic system (a Poincaré' map).

Note, since software metrics are numerous, many kinds of software trajectories describing a software product and process evolution can be constructed, including multidimensional trajectories. For

example a trajectory whose points consist of two measurements – churn (i.e., the velocity of software process) and cyclomatic complexity – can be constructed in an attempt to assess the system's complexity evolution. Current STA implementation is unable to work with multidimensional data type. Nevertheless, it can be adopted and used for multidimensional data, as I shall discuss in the Section 5.4 that is concerned with the future work.

Software Trajectory Analysis was proposed as a paradigm (i.e., a model) which, potentially, enables the extraction of *meaningful* patterns from software trajectories [191]. As a particular criterion for the pattern meaningfulness, its association with recurrent behaviors is considered.

Note, that STA was envisioned as a part of a larger, already existing system, called Hackystat [74], which provides an automation for sophisticated software process and product measurements. However due to a number of reasons, discussed throughout this chapter and in particular in Section 4.2.2, STA evolved into a stand alone tool which nevertheless can be plugged into Hackystat without any significant effort, thanks to the generality of the implementation.

4.1.1 Software Trajectory Analysis implementation

Discussed in this dissertation STA is implemented in Java and relies on a number of auxiliary libraries which aid in data collection [192], storage [193], and analysis [94]. STA also relies on the relational database engine which aids in data indexing and software trajectories construction.

STA does not have a single universal implementation. Currently, there exist three implementations customized for a particular case study (discussed in Sections 4.3.1, 4.3.2, and 4.3.3). This is due to the interactive nature of data mining, where a number of data and problem-specific abstraction and aggregation steps need to be performed sequentially in order to extract the knowledge. Typically, a project-specific STA implementation consists of a number of executable modules that need to be run sequentially in order to collect software artifacts, transform them into measurements, and to load these into the database. Similarly there are executable modules whose purpose is to extract and to analyze software trajectories. Therefore, in the following sections I shall discuss STA at the abstract level pointing out its specificity and limitations.

4.1.1.1 STA is generic

The SAX-VSM algorithm on which STA relies for patterns discovery, does not require the user to specify any baseline thresholds when performing analyses. The system is capable to discover class-characteristic software trajectory patterns directly from the provided data. All discussed in this chapter case-studies, namely the Android OS and PostgreSQL release patterns discovery, PostgreSQL maintenance pattern discovery, and StackOverflow user pattern discovery, are built upon this feature.

In addition to the class-characteristic patterns discovery, SAX-VSM ranks discovered patterns by their class-characteristic power – the property which I relate to interestingness and meaningfulness. The adequacy of this relation is examined in all three case studies.

As it is, STA can be applied to *two or more* sets of software trajectories that represent logical classes, such as different projects, teams, developers, etc. Alternatively, software trajectory classes can be defined as those generated by the same entity but within distinct, non-overlapping time intervals. These intervals can be associated with specific processes (such as software release or Scrum spike) or other external and internal constraints. This approach is used in the Android OS and PostgreSQL case studies, where software trajectory classes are defined by using different time intervals while the software trajectory-generating entities are staying the same.

Yet another STA specificity is that it does not place any constraints on the form of a provided dataset. Specifically, by its design, it is robust to any kind of asymmetry among volumes of the input classes and unequal lengths of software trajectories within and among the classes. For example, in PostgreSQL case study, software trajectory lengths varied from few dozens to few hundreds of points within a class.

Finally note that built upon the core Information Retrieval algorithm that is Vector Space Model, STA can be finely tuned in many ways in order to achieve the goal. Among other refinements are various weighting scheme (shown in Table 3.3), characteristic vector improvement through the relevance feedback [7], and other tuning techniques [194].

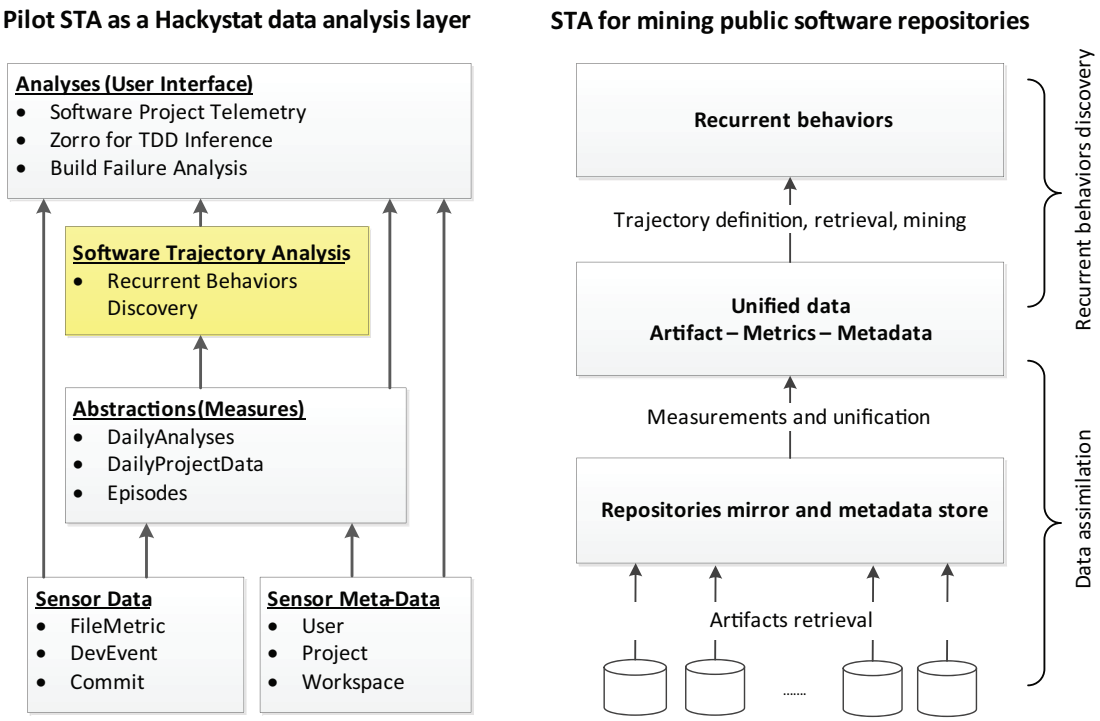


Figure 4.1: A schematic overview of the very first and the latest STA implementations. Note the STA evolution from a thin layer embedded into the larger system relying on external data assimilation and processing mechanisms to an end-to-end generic solution for software artifact measurements analysis.

4.1.1.2 STA is a two-components system

In order to enhance STA's generality, and to reduce the overall system complexity, a decision was made to decouple the data assimilation and the data analysis components using a relational database. This solution, shown at the right panel of Figure 4.1, allowed to successfully cope with a variety of data formats from numerous software process management and configuration systems since the internal STA data format stays unchanged for all upstream analyses allowing for interactive and efficient trajectory classes definition and their characteristic pattern discovery.

While the database schema supporting this design varies from project to project accommodating specific data types, it is usually simple as it only contains few tables that store artifact measurements and entities that facilitate their partitioning, such as user and project records. As an example, consider the database schema used in the Android OS case study shown in Figure 4.2. There, tables `change_target` and `android_change` contain information about source-code change events

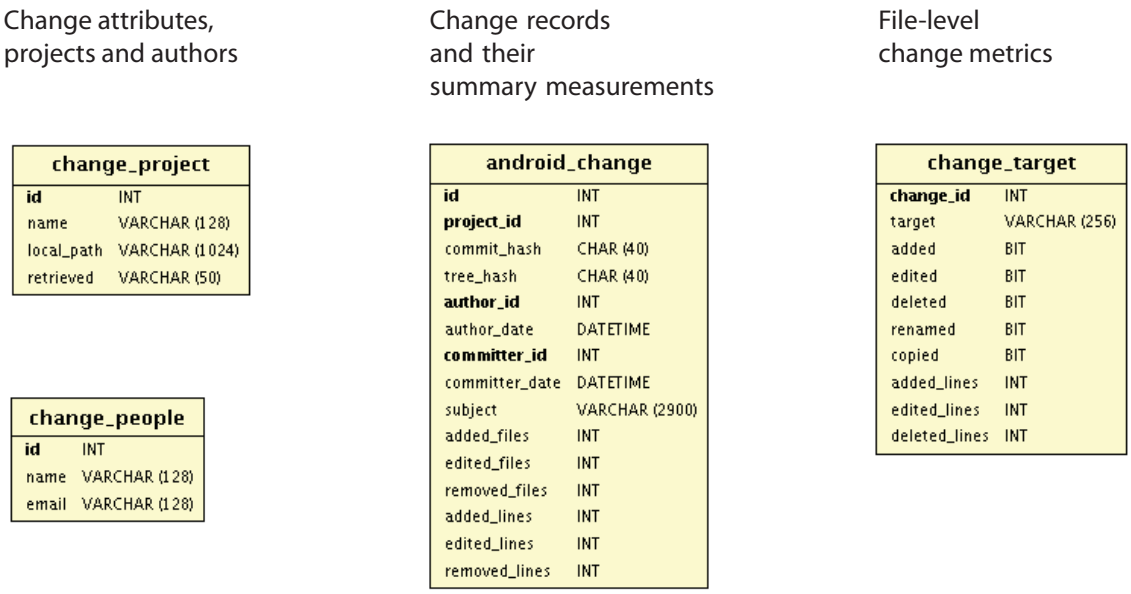


Figure 4.2: An example of the STA database schema used in Android OS case study which targets the discovery of recurrent behaviors from a history of software change records. As shown, the schema can be divided into three structural components where the change records and their summary measurements constitute the main table (middle). These are complemented by the information about the atomic changes (right). The tables enumerating sub-projects and committers (left) are used for the data partitioning, i.e., software trajectories construction.

and their measurements, while other tables, namely `change_people` and `change_project`, allow for the efficient software trajectories construction when using a simple SQL `SELECT` query. For example the following query retrieves a software trajectory for the Android OS contributor:

```
SELECT sum(c.added_lines) ‘value‘,
DATE_FORMAT(c.author_date, "%Y-%m-%d") ‘date‘ from OMAP.change c
where c.author_id=174 and c.project_id=1
AND c.author_date BETWEEN "2012-03-26" AND "2012-04-01"
GROUP by ‘date‘ order by ‘date‘;
```

Currently, STA relies on MySQL database server [195], but any other relational database engine can be used since all database communications are performed through an object-relational mapper called MyBATIS [193] which can be re-configured independently from STA source code.

4.1.1.3 STA limitations

There are two major limitations of Software Trajectory Analysis that are associated with its current implementation.

The first limitation is that the two or more classes analysis paradigm is not suitable for the study of a single trajectory or a single class of trajectories. While recently I have proposed a solution that enables the discovery of recurrent patterns from a single time series that is built upon symbolic discretization, grammatical inference, and the resulting grammar complexity analysis [196], it is not discussed in this dissertation as it is not yet evaluated.

The second limitation is that while it is asserted that the application of STA to two or more classes of software trajectories guarantees (by design) to yield a ranked lists of class-characteristic patterns, where recurrent patterns shall be ranked as the most important, *it may fail to do so*. Such STA behavior is well understood and is directly linked to the specificity of the input data: if it contains patterns that are similar across classes under analysis, they are dismissed from the resulting list by the **idf** component of VSM weighting schema as shown in Equation (3.10). In addition, when working with *only two* classes of trajectories, due to this phenomena STA reports only patterns that appeared in a single class, which is a very conservative approach. For example, consider that a SAX word accounts for 50% of all words in Class 1 and has been observed only once in Class 2: currently, in spite of the apparently high class-characteristic potential of the word, it will be discarded since its **idf** $= 0$ and consequently **tf**$*$**idf** $= 0$.

In order to handle this two-class issue, I employ a technique that is based on the re-labeling of samples and clustering, as discussed in the Android OS and PostgreSQL case studies. For this, all the trajectories are re-labeled with unique names and treated with SAX-VSM at first; next, the k-means clustering procedure (where k is set to 2) is applied; finally, the clusters are labeled by the members voting and their centroids are considered as class-characteristic pattern vectors. As I shall show, this approach demonstrates a promising performance. Note that this solution is not new and was pointed out before in a number of studies [197] [7] [194].

4.2 STA Pilot studies

Probably the most valuable in terms of insight gained into the problem of recurrent behaviors discovery from software artifacts were two exploratory studies conducted within the feasibility study phase of my research work. While the first study confirmed the possibility of recurrent behaviors discovery from artifact measurements, the SAX-VSM algorithm was developed and evaluated throughout the second study.

4.2.1 Feasibility study 1: mining Hackystat software telemetry streams

In order to investigate the feasibility of recurrent behaviors discovery from software process measurements, I have conducted a pilot study consisting of two experiments. The first experiment was based on the software telemetry streams discretization with SAX [4], patterns extraction, and their frequency-based analysis. The second experiment was based on the association rule mining algorithm application to series of software development events.

Software telemetry is a data type data that is generated by the Hackystat [75], which is an in-process software engineering measurement and analysis system. Software telemetry is collected with automation and is characterized by high consistency that enables unprecedented insight into performed processes, as I have already discussed in Section 1.5.2. Effectively, by offering the efficient data collection, storage, retrieval mechanisms, and most importantly the consistent, fine-grained data, Hackystat provided an ideal testbed for the STA feasibility study.

The data used in the study was collected from the development and deployment environments utilized by students participating in the Software Engineering class. The dataset represents Hackystat metrics collected during sixty days of a classroom project by eight students.

An overview of the pilot Hackystat-based STA targeting recurrent behaviors discovery is shown at the left panel of Figure 4.1. As mentioned, the first experiment was based on two analytical techniques: the discretization of time series with SAX, that effectively translates real-valued telemetry streams into strings, and the occurrence frequency (i.e., support) -based discovery of recurrent patterns that is similar to that formalized and discussed later by Lin et al. in [91].

As I have shown in [93] this approach demonstrated the feasibility of recurrent behaviors dis-

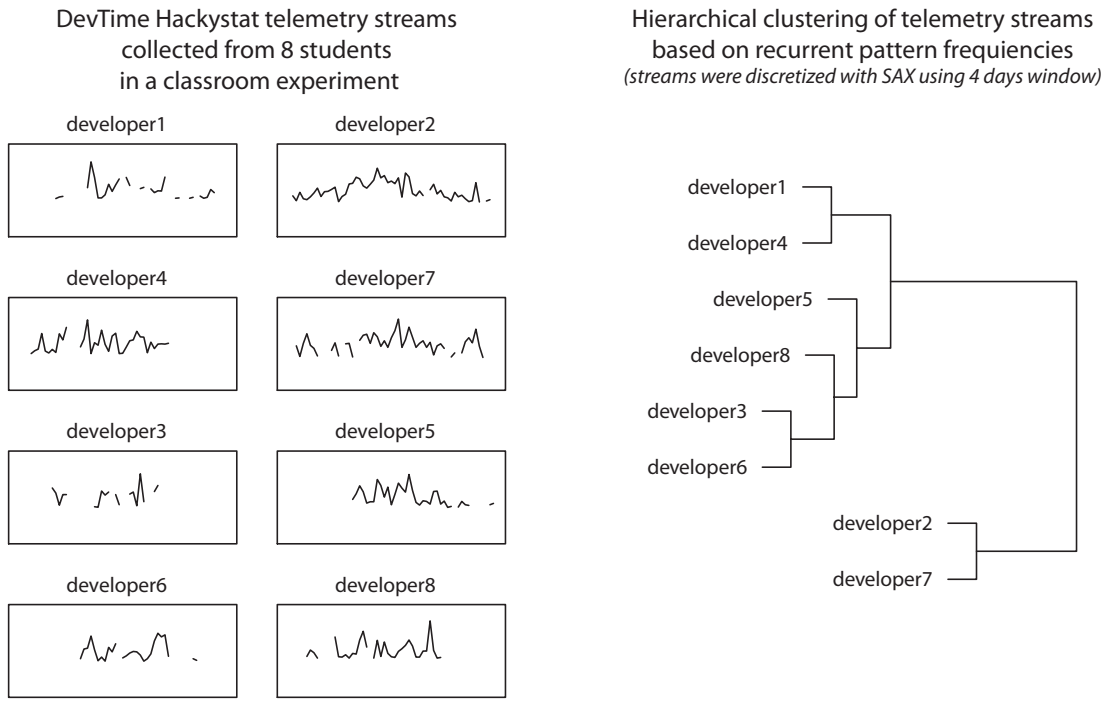


Figure 4.3: Results of the pilot STA study. The left panel shows eight software trajectories that are Hackystat telemetry streams corresponding to development effort [74] collected from eight developers in the course of two months. The right panel shows a hierarchical clustering of developers by the comparison of trajectory-corresponding sets of recurrent patterns discovered with SAX discretization [92]. Note two distinct groups discovered by clustering: the one that contains consistent trajectories (developers #2 and #7) and the one with less consistent trajectories.

covery by mining of frequently occurring symbolic patterns, i.e., time series motifs [92]. Consider an example of recurrent behaviors discovery shown in Figure 4.3, where software trajectories reflecting the development effort measurements shown on the left and their clustering based on the Euclidean distance between vectors of symbolic patterns occurrence frequencies shown on the right. Clearly, the hierarchical clustering process partitioned the set of trajectories separating two developers (#2 and #7) from the rest. Further investigation of the data revealed that these two developers demonstrated the most consistent development behavior (when discretized by 4 days window) as they spent considerable amounts of time working on the project almost daily whereas the rest of the study participants did not. Thus, the results of STA analysis were found consistent with the ground truth.

In addition to indicating the feasibility of automated recurrent behaviors discovery through the

analysis of discretized measurements, the experience with pilot system highlighted a number of issues. It was found that the major issue threatening the external validity of study, was the small scale of class-room experimentation that simply did not provide an adequate and generalizable coverage of the studied phenomena. For example, it is possible that in the above experiment some of the developers characterized by "inconsistent behavior" may simply had their Hackystat sensors misconfigured or malfunctioning, which is difficult to recognize automatically. The second significant issue identified through experimentation was the problem of discretization algorithm parameters selection – they have to be defined as the input, but their proper values are non-intuitive and often difficult to guess.

The second experiment investigated the applicability of an association rule mining algorithm called Apriori [198] to the the stream of development event records collected by Hackystat. As I have shown in [191], this approach also demonstrated a satisfactory performance. However, since it is impossible to recover the development events from public software artifacts, as discussed in the Section 2.3, this workflow has not been used in the following STA implementations.

4.2.2 Feasibility study 2: mining public software repositories

Following lessons learned during the pilot study and the feedback collected through its discussion [93], the decision has been made to explore the feasibility of recurrent behaviors discovery from software trajectories constructed by measuring public software artifacts. The chief reason behind that decision is an attempt to increase the generality and significance of findings by addressing all of the essential characteristics for empirical studies based on mining software artifacts proposed by Gasser et al. [117]: (1) they must reflect a real-life phenomena, (2) provide adequate phenomena coverage, (3) examine representative levels of variance, (4) demonstrate an adequate level of statistical significance, (5) provide results that are comparable across projects, (6) be reproducible.

Unfortunately, due to much coarser granularity and inconsistency of software trajectories constructed by measuring public software artifacts, the original approach to data analysis based on frequency of observed patterns failed, and an additional study of time series mining techniques has been conducted using 2012 MSR challenge data [113] from the Android OS repository. Discovery

of recurrent behaviors associated with the *software release pattern* was set as the study's goal.

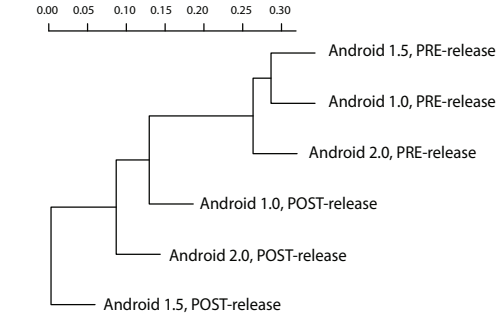
4.2.2.1 Software release pattern

Previously, in the software engineering literature, it has been proposed, discussed, and shown that different software development cycles, and in particular the software implementation, release, and maintenance, impose various constraints on software processes [199] [200] [201] [202]. Later, Hindle et al. in [61] have shown that it is possible to discover the software release pattern via partitioning of software process artifacts. The authors aggregated change summaries using STDB notation (S for source, T for test, B for build, D for documentation) and have shown that the behavior of STDB summaries changes around the software release.

4.2.2.2 Software release pattern discovery with STA

Taking into account the release pattern significance and the previous experience in its discovery through analysis of public software artifacts, I have explored the possibility of software release-characteristic recurrent behaviors discovery using STA and Android OS data. By experimenting with a number of time series transformation, discretization, and aggregation techniques, as well as with various distance functions and ranking schema, I found that the common in Information Retrieval (IR) toolkit called Vector Space Model (VSM) [171] that is based on **tf**$*$**idf** ranking schema and Cosine similarity, demonstrated a satisfactory performance. Specifically, as I have shown in [96], STA based on the discretization with SAX [92] and mining with VSM [171],was found capable to discover characteristic behaviors in pre- and post- release software trajectories constructed out of *New Lines of Code* change record measurements by following the clustering methodology discussed in Section 3.8.

Consider an example shown in Figure 4.4 for two classes of software trajectories that reflect pre- and post- release dynamics in counts of *New Lines of Code* in the Android OS OMAP repository. The left panel of the figure shows that it is possible to cluster characteristic behaviors corresponding to different time intervals where pre- and post- release behaviors are clearly separated. The right panel shows that by using pre- and post- release clusters centroids it is also possible to build a "soft-



Classification of other OMAP kernel releases using pre- and post- release cluster centroids

Release	Classification
1.6-pre	misclassified
1.6-post	OK
2.2-pre	OK
2.2-post	OK
1.1-pre	OK
1.1-post	OK
2.3-pre	OK
2.3-post	OK

Release	Classification
beta-pre	OK
beta-post	OK
2.0.1-pre	OK
2.0.1-post	misclassified
2.1-pre	OK
2.1-post	OK
2.2.1-pre	OK
2.2.1-post	misclassified

Figure 4.4: An example of the discovery of recurrent patterns in software trajectories constructed by measuring Android OS repository source code change artifacts. The left panel shows the hierarchical clustering of pre- and post-release temporal interval-corresponding software trajectories based on the Cosine similarity applied to ranked vectors of discovered characteristic patterns. The right panel shows the result of a cross-validation experiment where other pre- and post-release software trajectories were classified by computing their NN similarity with previously discovered patterns.

ware release behavior classifier" that properly assigns the majority of test intervals collected from other (not used for training) time intervals surrounding releases. The latter validates the discovered recurrent patterns characteristic capacity and the overall correctness of approach.

To combat the lack of Android software repositories internal and external connectivity and the heterogeneity of data formats – also a common issues in the MSR field – in this STA implementation I had followed state of the art MSR approaches for data integration [117] [101]. In particular, similarly to a previously developed solution called softChange [100], STA mirrors repositories and builds its own data storage facility by using a relational database engine as shown in Figure 2.3.

Note, that similarly to the pilot implementation, the experience with second STA highlighted the same problem of parameters selection. Moreover, this issue became even more significant since the proposed methodology was found sensitive to parameters selection. In order to address this issue, I have explored a parameters optimization scheme and implemented a DIRECT algorithm-based approach [2] that aids in parameters selection – the project that essentially led to SAX-VSM development.

4.3 STA 2.0 Case studies

STA 2.0 is the most current implementation of proposed in this dissertation framework targeting the discovery of recurrent behaviors from software trajectories. It addresses all of the previously identified weaknesses and embeds all the effective solutions found throughout my exploratory studies.

In particular, STA 2.0 is built upon the SAX-VSM algorithm including the DIRECT-based parameters optimization schema, and has a layered design where the trajectory analysis part is decoupled from the data assimilation part by a relational database.

In the next sections I shall discuss three case studies examining the applicability and performance of STA 2.0:

- The Android OS software release characteristic behaviors discovery.
- The PostgreSQL software maintenance and software release characteristic behaviors discovery.
- The StackOverflow top ranked users characteristic behaviors discovery.

4.3.1 Case Study 1: Android OS software release recurrent behaviors discovery

As discussed above in Section 4.2.2, during the second pilot study I have used a time interval fixed to one week and a specific subset of users having corporate e-mails, which, in my opinion, should have followed some distinguishable software development pattern.

While this approach is logical, and is suitable for a feasibility study, it puts unreasonably strict constraints on the input data and creates a significant internal validity threat since STA only considers and reports week-long behaviors characterizing a limited group of people, which may not characterize the performed software processes adequately. This limitation was also pointed out by the reviewers of the describing the pilot publication [96].

Addressing these limitations, I have designed and performed a new experiment targeting the discovery of the Android OS software release characteristic behaviors when accounting for **all** available information.

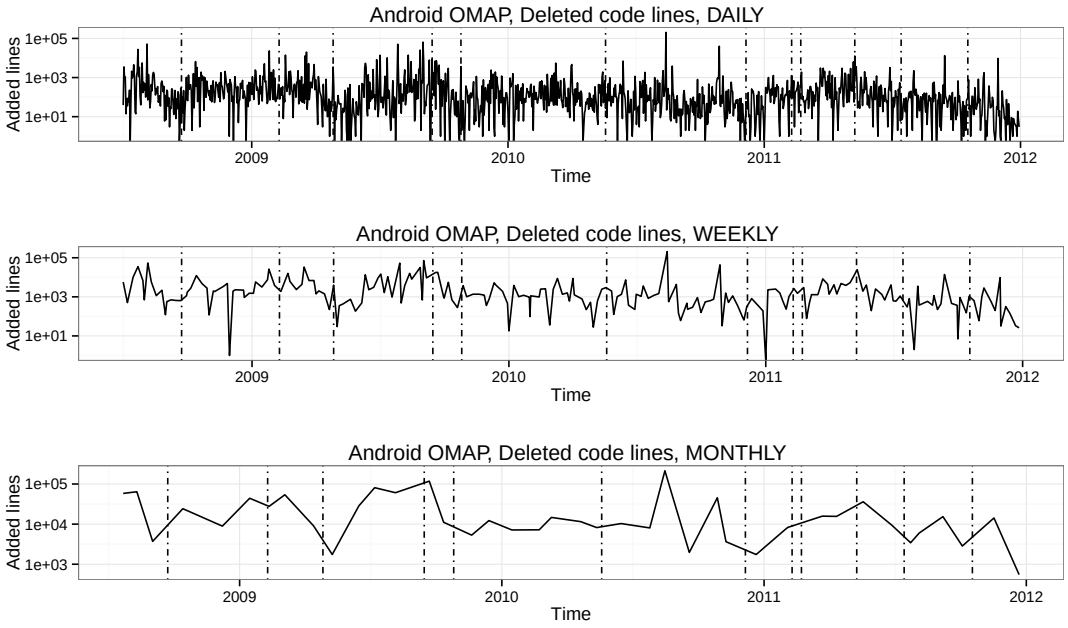


Figure 4.5: The dynamics of *Deleted LOC* measurements throughout Android OS kernel OMAP evolution. The 12 software release dates shown by vertical lines.

4.3.1.1 Android OS dataset

The data for this study was collected by using the STA toolkit. First, the Android OS kernel OMAP repository was mirrored in order to avoid the network latency. Next, software change records were measured by STA using the mirror and populated into a dedicated database, whose schema is shown in Figure 4.1. This enabled an efficient measurements indexing and instant software trajectory construction, as explained in Section 4.1.1.2.

Note that while the Android OS dataset contains 102,602 change records authored by 7,103 authors, only 501 users were recognized by STA as active committers. This observation indirectly supports the exploratory study hypothesis that the studied project development is likely to follow some form of software process where only "trusted developers" (*most of them having a corporate email address*) are allowed to write changes into the repository.

Table 4.1: Counts of pre- and post- release trajectories corresponding to the *Deleted LOC* dynamics per author and time interval within Android OS kernel OMAP project.

Id	Release name	API level	Date	Pre-release *Deleted LOC* trajectories count	Post-release *Deleted LOC* trajectories count
1	Android 1.0	1	2008-09-23	266	381
2	Android 1.1	2	2009-02-09	302	342
3	Android 1.5	3	2009-04-27	330	177
4	Android 1.6	4	2009-09-15	214	390
5	Android 2.0	5	2009-10-26	252	255
6	Android 2.2	8	2010-05-20	292	368
7	Android 2.3	9	2010-12-06	209	203
8	Android 2.3.3	10	2011-02-09	364	274
9	Android 3.0	11	2011-02-22	308	341
10	Android 3.1	12	2011-05-10	324	416
11	Android 3.2	13	2011-07-15	314	238
12	Android 4.0	14	2011-10-18	186	194

4.3.1.2 Study design

I have used three types of measurements in this study, that is (i) *New LOC*, (ii) *Edited LOC*, and (iii) *Deleted LOC* considering 12 major releases (Android API levels 1–14, excluding API level 6 and 7 which were the minor improvements [203]) indicated in Table 4.1. The Lines Of Code (LOC) measurements were used since they represent a programmer's raw output and has been shown to reflect the size and complexity of software system along with the productivity of programmers [204] [205] [206]. Since in experiments software trajectories comprised of *Deleted LOC* measurements were found as having the most class-characteristic power, this measurement dynamics throughout the project history is shown in Figure 4.5 with variable granularity. Note, that the total daily values within this data stream vary from zero to few thousands while the average activity slowly decreases.

For each of the release dates, the release week was determined and excluded from analyses. Intervals equal to four weeks preceding, and four weeks succeeding the release week were extracted and used in the study while named as *pre-release* and *post-release* intervals respectively. For each contributor that authored a change record resulted in source code lines measurements change within pre- and post- release intervals, software trajectories were constructed. The total amount of trajectories within pre- and post-release intervals in *Deleted LOC* measurements is shown in Table 4.1.

Pre-release centroid	
pattern	weight
ebbbebbbbbbb	0.1748588272
bbbbbcbbbebb	0.1083403654
bbbbbbbdebbb	0.0901908199
bbbbbbbbdebb	0.0901908199
bbbbbbdebbbb	0.0901908199
...	...

Post-release centroid	
pattern	weight
edbbbbbbbbbb	0.1995655982
bbbbbebbcbbb	0.1533399084
bbbbbbebbcbb	0.1533399084
bbbbbbbebbcb	0.1533399084
bbbbbbbbebbc	0.1533399084
...	...

Table 4.2: An excerpt from pre- and post-release class-characteristic pattern vectors obtained by mining the *Deleted LOC* trajectories in Android OS case study. The total size of each vector is 622 weighted patterns.

Similar to that in the feasibility study, I have used three random software releases in order to discover pre- and post-release class-characteristic patterns. First, in order to retain more class-characteristic patterns (addressing the limitation discussed in section 4.1.1.3), trajectories labeled by two labels (pre- and post- release) were relabeled at first by assigning them to three pairs of pre- and post- release software trajectory classes labeled as *pre-1*, *pre-2*, *pre-3*, and *post-1*, *post-2*, and *post-3* respectfully. At the second step, SAX-VSM was applied to these six classes and the optimal parameters set was determined with the DIRECT optimization scheme (Section 3.5). At the third step, software trajectories from each class were discretized into a bag of words with SAX using optimal parameters and **tf**$*$**idf** statistics was computed. Finally, the resulting weight vectors were clustered using SAX-VSM implementation of spherical k-Means clustering (Section 3.8) with k=2 and the resulting cluster centroids, corresponding to pre- and post-release clusters, were extracted. These centroids were used in the validation step as vectors comprised of class-characteristic patterns. An example of these vectors is shown in Table 4.2.

The class-characteristic vectors computed at previous step were evaluated for class-characteristic power using cross validation. For this, a SAX-VSM classifier was constructed and its accuracy was determined by classifying pre- and post- release trajectories corresponding to all software releases under analysis.

Software metric	Train releases	Parameters	Accuracy	Note
added code lines	1,3,5	18,7,12	54.00%	biased towards post-
added code lines	4,6,9	15,15,5	58.33%	biased towards post-
added code lines	5,8,11	12,10,10	66.66%	biased towards pre-
added code lines	1,6,12	28,5,14	66.66%	biased towards pre-
edited lines	1,3,5	24,10,4	62.50%	biased towards post-
edited lines	4,6,9	24,5,12	58.33%	biased towards post-
edited lines	5,8,11	22,7,7	62.50%	biased towards pre-
edited lines	1,6,12	18,8,7	58.33%	biased towards pre-
deleted lines	1,3,5	24,10,4	58.44%	biased towards pre-
deleted lines	4,6,9	12,12,5	**75.00**%	
deleted lines	5,8,11	24,5,7	61.50%	biased towards post-
deleted lines	1,6,12	24,5,11	62.50%	biased towards pre-

Table 4.3: Statistics for a number of software release classifiers built using the Android OS data. As shown, a typical classifier for post- and pre- release behaviors based on LOC change measurements achieves an accuracy above 60%. The best performing classifier demonstrated 75% accuracy and was trained on using the deleted lines of code measurements corresponding to API level releases {4,6,9}.

4.3.1.3 Results

The outlined above procedure was applied to 4 random samples each of which consists of 3 releases (*Train releases* in Table 4.3) using three types of software trajectories (*Software metric* in Table 4.3). The accuracy of resulting classifiers is shown in Table 4.3. As shown, the best performing classifier was built using the intervals corresponding to the set of releases {4,6,9} (i.e., Android OS API Levels 4, 8, and 11) and the *Deleted LOC* measurements.

Table 4.4 shows details of the classification with the best performing classifier. As shown, it is slightly biased towards post-release. The first 5 class-characteristic patterns for both classes are shown in Table 4.2, while examples of software trajectories containing these are shown in Figure 4.6.

4.3.1.4 Discussion

Quite intriguing and unexpected, the best pre- and post-release class-characteristic patterns were discovered in software trajectories comprised of the *Deleted LOC* measurements. These were found using the discretization parameters of sliding window 12, PAA 12, and alphabet of the size 5, i.e.,

Class	pre-cosine	post-cosine	classification result
pre-1	0.0112	0.0076	ok
pre-2	0.0073	0.0095	miscl.
pre-3	0.0108	0.0083	ok
pre-4	0.0223	0.0066	ok
pre-5	0.0093	0.0143	miscl.
pre-6	0.0061	0.0143	miscl.
pre-7	0.0083	0.0088	miscl.
pre-8	0.0120	0.0107	ok
pre-9	0.0186	0.0076	ok
pre-10	0.0095	0.0085	ok
pre-11	0.0128	0.0088	ok
pre-12	0.0115	0.0091	ok

Class	pre-cosine	post-cosine	classification result
post-1	0.0088	0.0128	ok
post-2	0.0098	0.0074	miscl.
post-3	0.0056	0.0081	ok
post-4	0.0077	0.0175	ok
post-5	0.0049	0.0058	ok
post-6	0.0055	0.0144	ok
post-7	0.0083	0.0100	ok
post-8	0.0100	0.0104	ok
post-9	0.0189	0.0068	miscl.
post-10	0.0116	0.0128	ok
post-11	0.0087	0.0103	ok
post-12	0.0071	0.0072	ok

Table 4.4: The classification results for the Andriod OS release classifier built using *Deleted LOC* software trajectory. Higher cosine value corresponds to smaller angle and is better. Overall, this classifier demonstrated an accuracy of 75%.

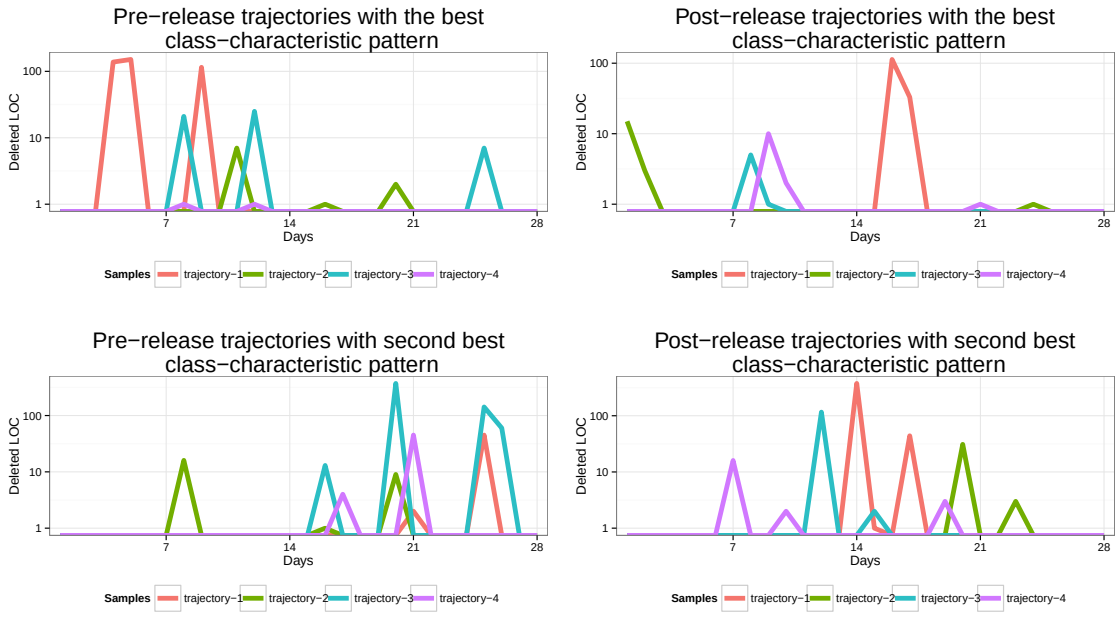


Figure 4.6: Examples of the recurrent behaviors discovered using *Deleted LOC* software trajectories from Android OS dataset.

by converting normalized daily measurement into a letter taken from an alphabet of the size 5.

The patterns shown in Figure 4.6 reveal that the best class-characteristic behavior for pre-release

class when accounting to the *Deleted LOC* measurements is to perform two deletions of approximately equal volume separated by three days and followed by a week of inactivity, whereas the best class-characteristic behavior for the post-release class is to perform decreasing in volume deletions during two consecutive days followed by 10 days of inactivity. Second best class-characteristic patterns were also found to follow a similar pattern – different in volume source code lines deletion events separated/followed by a time interval.

Overall, the discovered patterns are impossible to translate into a sensible description without discussion with developers. Unfortunately, despite of my effort, I was not able to communicate with key contributors from the Android OS kernel OMAP team.

Through my own investigation of commit messages and source code files corresponding to deletion events of the best pre-release patterns, I have found that majority of them correspond to a normal software development cycle where the changes were staged, reviewed, and signed off by the project managers. What was interesting however, is that many of the deletion events were reflecting the code clean-up from Linux artifacts (Android OS is based on the Linux kernel), such as SCSI modules, or other platform hardware-related code, therefore, since observed among many trajectories, they may reflect a systematic Android OS release-related activities.

4.3.2 Case Study 2: PostgreSQL software maintenance and software release recurrent behaviors discovery

Similar to the previous case study, I have explored the possibility of recurrent behaviors discovery from software trajectories that were constructed by measuring software change artifacts from PostgreSQL public software repository.

PostgreSQL is an open-source database developed by the PostgreSQL Global Development Group consisting of a number of volunteers employed and supervised by companies such as Red Hat and EnterpriseDB [207]. It has a large number of extensions written by contributors and is available for many platforms including Linux, FreeBSD, Solaris, Microsoft Windows and Mac OS X.

One of the particular characteristics of the PostgreSQL software development process is its regular CommitFest events [208]. As PostgreSQL team explains it, a CommitFest (CF) event is a "*periodic break to PostgreSQL development that focuses on patch review and commit rather than new development*" – a description that allows to classify it as a *maintenance activity* whose purpose is to promptly review and to respond with a feedback to development community without waiting for a major release. Contributors are encouraged by the core development team to submit patches into the development mailing list. Within a CF event, these patches are reviewed, tested, and the decision for a final review and commit is made. Typically, CFs tend to run for one month with a one month gap between them, however, when the core team is busy with a PostgreSQL major release, there may be several months without CF events followed by a ReviewFest (RF), which helps to pre-organize patches, and a CF .

Up to the data retrieval date, 18 CF events were held. Typically, after reviewing and testing of a patch submitted for CF, developers assign it to one of the categories: "Needs Review", "Ready for Commit", "Committed", "Returned with Feedback", or "Rejected". While the very first CF event dealt with 66 patches, from which 37 were committed, the latest CF event dealt with 108 patches in the review queue out of which 7 were marked for additional review, 14 as ready to commit, 36 were committed, and 42 were returned with a feedback.

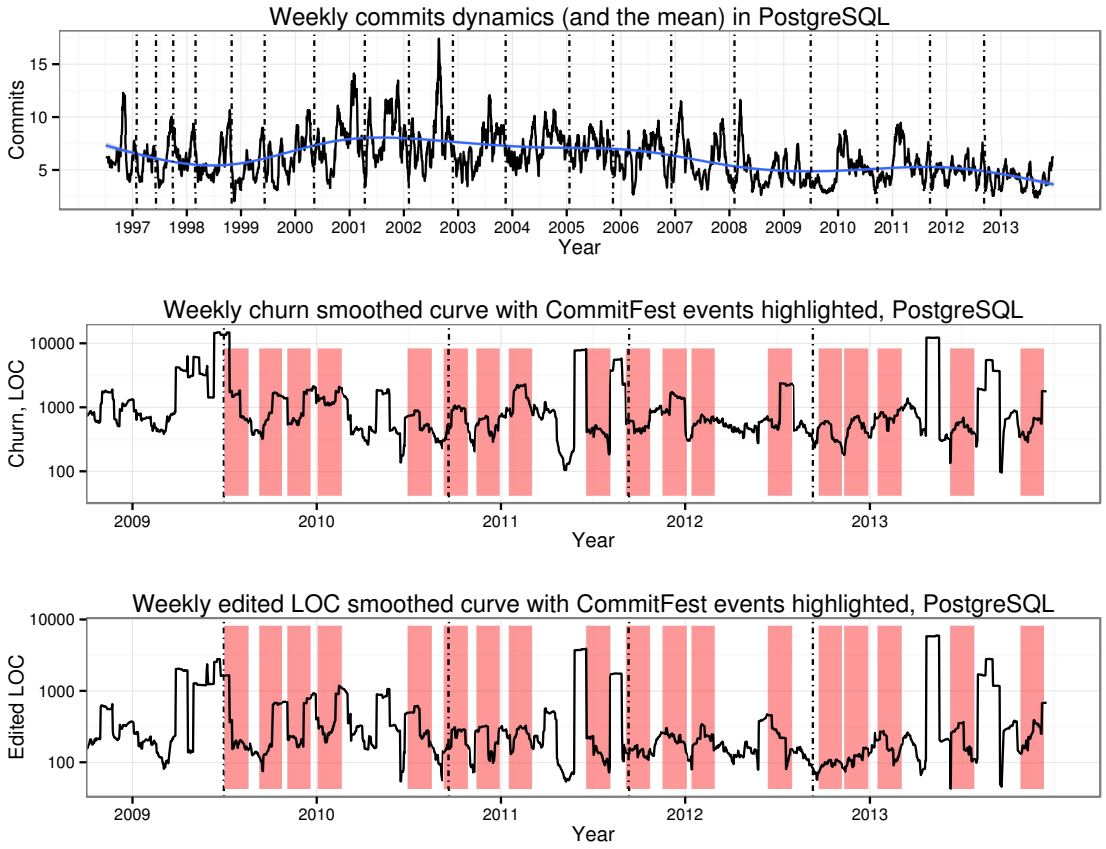


Figure 4.7: PostgreSQL source code change evolution. The top panel shows dynamics of the weekly commits into PostgreSQL repository, the middle panel shows *Churn*, and the bottom panel shows *Edited LOC* measurements dynamics throughout the analyzed Commit Fest events. Dotted vertical lines show the release dates.

4.3.2.1 PostgreSQL dataset

Similar to Android OS study, the PostgreSQL data was collected by using STA data assimilation toolkit and stored in the same database. The dataset consists of 35,890 change records authored by 38 authors. The overall commit activity shown at the top panel of Figure 4.7 indicates that the project has been active throughout the years. The middle and bottom panels of Figure 4.7 show *Churn* and *Added LOC* aggregated software trajectories and Commit Fest events.

Commit Fest behaviors experiment		
Trajectory class	Discretization parameters	LOOCV accuracy
added LOC	6,5,8	72.22%
edited LOC	14,5,5	**75.00**%
deleted LOC	8,6,10	**75.00**%
added files	12,8,5	65.71%
edited files	12,4,11	66.67%
deleted files	27,7,3	55.17%

Software Release behaviors experiment		
Trajectory class	Discretization parameters	LOOCV accuracy
added LOC	14,5,7	**80.56**%
edited LOC	5,5,14	75.00%
deleted LOC	10,5,11	72.22%
added files	16,4,10	64.71%
edited files	6,4,7	**80.56**%
deleted files	18,5,12	56.25%

Table 4.5: The Leave One Out Cross Validation results for PostgreSQL aggregated software trajectories. The discretization parameters are ordered as the sequence of sliding window size, PAA size, Alphabet size.

4.3.2.2 Study design

Based on the PostgreSQL development team documentation of their software maintenance process called Commit Fest [208], the main goal of this study was to discover Commit Fest -characteristic recurrent behaviors. The secondary goal was to explore the software release pattern for 19 PostgreSQL releases from 6.0 dated by 1997-01-29 to 9.2 dated by 2012-09-10. The releases are shown in Figure 4.7.

In this study, since the average activity of individual contributors is quite sparse, I have used aggregated software trajectories which were constructed by measuring *all* change records without differentiating them by committers or authors. These aggregated trajectories, in turn, were cut into the pieces representing CF and non-CF software trajectories using stipulated in [208] dates. For example, for *Added LOC* measurements, a single software trajectory was constructed at first, then, its continuous intervals within Commit Fest intervals were extracted and labeled as CF-corresponding software trajectories, whereas the rest of continuous intervals was labeled as non-CF software trajectories. The pre- and post-release software trajectory classes were constructed in the similar fashion but by using four weeks preceding and four weeks succeeding the release week.

Overall there were 18 software trajectories constructed for Commit Fest class and 18 for non-Commit Fest class, whose length varied in a range from 27 to 183. In addition, 19 software trajectories for pre-Release and 19 software trajectories for post-Release were constructed, each of them spanning 28 days. Note, that the difference in trajectories length does not affect STA performance

as it was shown in Section 3.7.4.

For both, PostgreSQL Commit Fest and PostgreSQL Software Release experiments, a common Leave One Out Cross Validation (LOOCV) [209] evaluation was performed in order to estimate how accurately an STA-discovered predictive model (that is a VSM classifier based on the class-characteristic vectors) would perform in practice.

4.3.2.3 Results

The results of LOOCV experiments are shown in Table 4.5. Overall, similar to the Android OS study, it was found that a resulting characteristic-pattern based classifier performs with an accuracy above 60%. The best accuracy was achieved by using *Edited LOC* and *Deleted LOC* trajectories for Commit Fest study, whereas patterns from *Added LOC* and *Edited Files* software trajectories characterized the Software Release the best. Figures 4.8 and 4.9 show examples of patterns from both studies.

4.3.2.4 Discussion

First of all, note that in the PostgreSQL study, a typical classifier built upon class-characteristic behaviors discovered with STA achieved a comparable accuracy with that of Android OS study, while the best classifiers outperformed that of Android OS. Although this can be explained by differences in the experimental design (random training sample in Android OS and LOOCV in PostgreSQL), alternatively, the better result can be explained by a nature of used software trajectories – individual (Android OS) versus aggregated (PostgreSQL).

Second, note that in contrast to the Android OS study class-characteristic behaviors discovered in PostgreSQL study are easier to comprehend visually and to interpret. Both, the non-Commit Fest and pre-Software Release patterns are characterized by stretches of low activity interrupted by large in volume commits (the team focuses on the release and new development), whereas Commit Fest and post-Software Release trajectories are characterized by stretches of frequent, but moderate activity (team performs maintenance) – both findings are in accord with PostgreSQL process description [208].

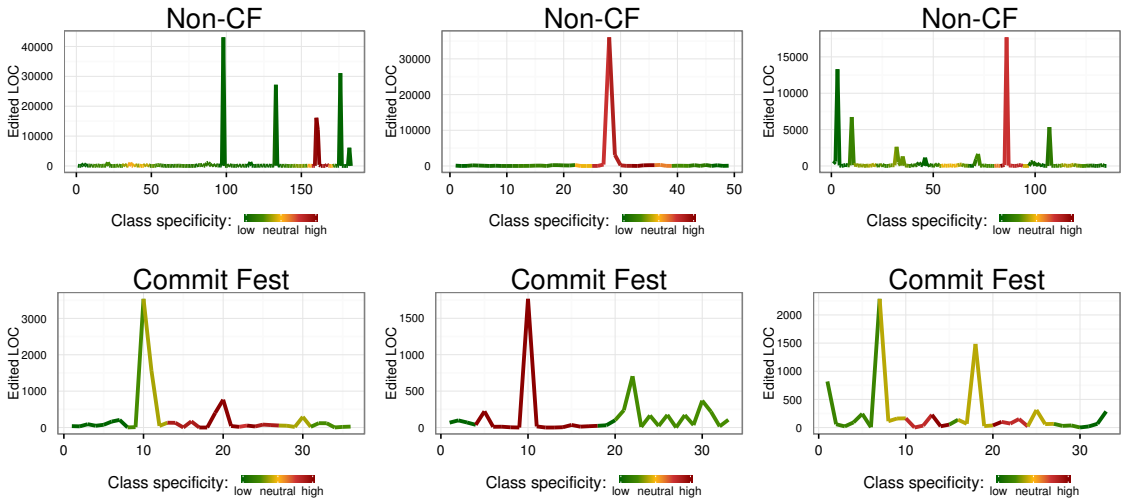


Figure 4.8: Examples of class-characteristic behaviors discovered by SAX-VSM in PostgreSQL Commit Fest experiments. Note, that the large commits surrounded by no-activity intervals are characteristic to the regular development, whereas smaller in the volume, frequent commits are characteristic to the Commit Fest -corresponding development intervals.

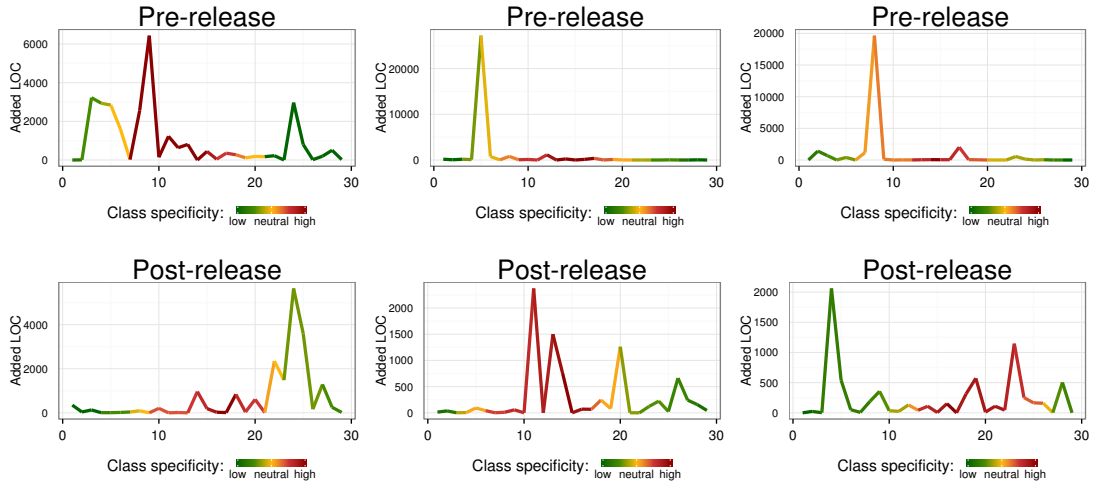


Figure 4.9: Examples of class-characteristic behaviors discovered by SAX-VSM in PostgreSQL Software Release experiments. Note, that relatively large commits followed by low activity are characteristic for pre-release intervals, whereas post-release development is characterized by frequent commits.

4.3.3 Case Study 3: mining user-characteristic behaviors in Stack Overflow data

Stack Overflow (SO) is a question and answer website created in 2008 that is primarily used by computer programmers. There, users are actively encouraged to participate in the community by creating public user profiles and engaging into discussions by asking good questions and providing relevant answers. As a form of gamification, this desirable user behavior is rewarded with a combination of a numerical score called reputation, and "badges" that implement a goals framework.

The reputation points are awarded when individual activities are performed, such as asking a good question, providing a good answer, or commenting. There is a hierarchy of badges, from the lowest "bronze badges", that are relatively common and easy to achieve, to "golden badges", that are awarded for long term dedication and recognition from the community. Overall, the reputation and badges are an estimate of how much the community trusts the user and how much valuable contribution she has provided. Naturally, these incentives lead users to attempt to achieve as much reputation and as many badges as possible to demonstrate their expertise and to gain respect in the community. Some of these badges can be awarded recurrently, which explains how user #22656, Jon Skeet, collected over 11,000 of these as per time of writing.

Several goals were set for this study. The first goal was to explore the STA applicability to the problem of discovery of characteristic recurrent behaviors from daily and weekly user activity patterns. The second goal was to explore the applicability of a popular bioinformatics tool called WebLogo [210], that creates graphical representations (logos) revealing significant patterns from a multiple sequence alignment. Since STA discovers recurrent patterns in the symbolic space, I have hypothesized that WebLogo figures shall allow summarizing numerous discovered patterns for visual comprehension. The third goal was to explore differences among the top SO users daily and weekly activity patterns in order to gain an insight into their productivity.

4.3.3.1 StackOverflow data

The data used in this study was obtained from the Stack Overflow public release dump that is dated by August 2012 and contains over four years of the website content evolution. The dataset contains information about the users, their comments, posts, and related activities, a subset of voting history,

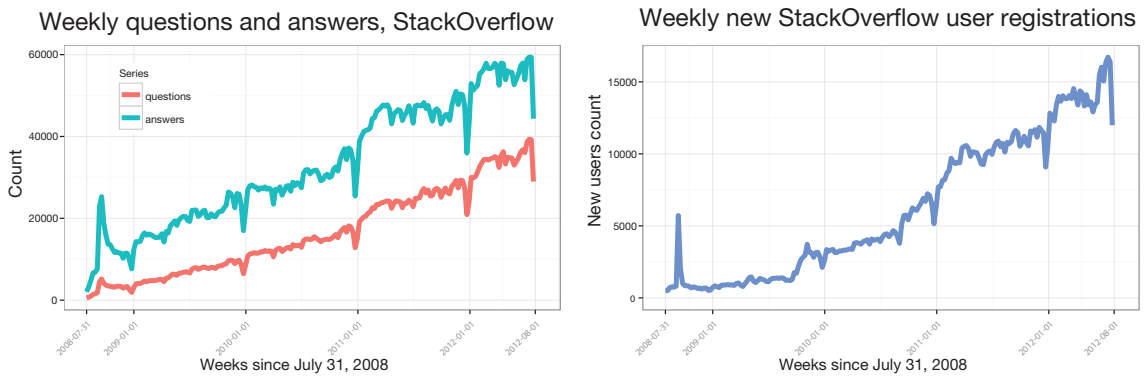


Figure 4.10: StackOverflow contributors activity weekly dynamics overview. Left panel shows the evolution of Questions and Answers, whereas the right panel shows the curve of new users registration.

User	Reputation	Answer acceptance rate	Daily trajectories before & after weighting		Weekly trajectories before & after weighting	
Jon Skeet	465166	60%	1401	318	199	113
Darin Dimitrov	343191	59%	1270	347	192	168
Marc Gravell	325797	52%	1384	525	197	153
BalusC	298811	66%	1002	329	142	83
Hans Passant	271982	59%	1165	355	177	148

Table 4.6: Descriptive statistics for the top StackOverflow users.

and records about awarded badges. Overall, the dataset accounts for 1.3M of users which created 10.4M of posts (3.5M of questions, 6.9M of answers), and 14M of comments. In addition, there is information about 28M of votes. The weekly dynamics of new Questions, Answers, and newly registered users is shown in Figure 4.10.

For the experimentation I have selected 5 top users whose summary is shown in Table 4.6. Note, that the top three users were active for the almost whole time span considered in this study.

4.3.3.2 Study design

In order to explore recurrent behaviors of five top StackOverflow users with STA, I have constructed software trajectories by summarizing amounts of user-created questions, answers, and comments per hour and per day. These were used to construct the daily and weekly activity trajectories. Next, each daily trajectory was discretized into a 8-letters string with SAX (i.e., by aggregating values for consecutive 3 hours) while each weekly trajectory was discretized into 7 letters string (a letter

Figure 4.11: A comparison of the activity pattern visualization techniques. Figures at left convey the most information by accounting for each hour and day, showing J. Skeet's increasing involvement over time. Plots at the middle convey the minimal amount of information by showing averaged summaries. Plots at right are made with WebLogo [210] using discretized with ABC-notation trajectories – these compactly convey the information about daily/weekly behaviors variance and frequency by the letter height; the longitudinal aspect is lost, however.

per day). For both discretization procedures I have used an alphabet of the size 3 whose letters (A, B, C) can be interpreted as ("*low*", "*medium*", and "*high*") activity levels. The intuition behind this ABC-coding schema is that it shall help to reveal the differences in users daily and weekly activity dynamics and, possibly, shed a light on the differences in their reputation score.

Within my dissertation proposal, and in the following work [93], I have discussed the possible use of Bioinformatics tools for the discovery and visualization of patterns extracted from discretized software trajectories. In this exploratory study, I have utilized a widely known visualization tool called WebLogo [210] that creates graphical representation of patterns found within a multiple sequence alignment. As pointed out by the authors, "*. . . sequence logos provide a precise description of sequences similarity and can rapidly reveal significant features of the alignment otherwise difficult to perceive*". Each logo generated by the tool consists of stacked letters, one stack for each position in the sequence. The overall height of each column indicates the sequence conservation at that position, while the height of symbols within the column reflects the relative frequency of the corresponding letter at that position.

Figure 4.11 shows a comparison of WebLogo figures with two other visualization techniques con-

Figure 4.12: WebLogo visualization of daily activity for top SO users. Here, letters (*A,B,C*) corresponds to (*low,medium,* and *high*) levels of activity. Note, that SAX-VSM pattern ranking process changed the effort distribution. The recurrent behaviors shown at logos were partially confirmed by respective SO users. The excluded behaviors represent a very common behavioral pattern [211]: the increasing activity levels from 9AM to 3PM and the decreasing activity levels from 3PM to 12AM.

veying the same information about Jon Skeet's behaviors: the rug plot, and the averaged curve. As shown, the logo provides less resolution than a rug plot, but much more than a curve, which makes it an acceptable visualization tool when accounting for internal symbolic information representation within STA. While WebLogo allows the user to specify palette of colors for each letter, in this study I have used the two colors scheme for simplicity and in order to contrast high intensity intervals.

4.3.3.3 Results

The results of STA and WebLogo application to StackOverflow data are shown in Figure 4.12 for daily patterns and in Figure 4.13 for weekly patterns. In each figure I also compare the WebLogo-created logos for user-characteristic behaviors before and after applying SAX-VSM ranking. Note, that the ranking changes not only the amount of observed patterns (Table 4.6) but the activity levels

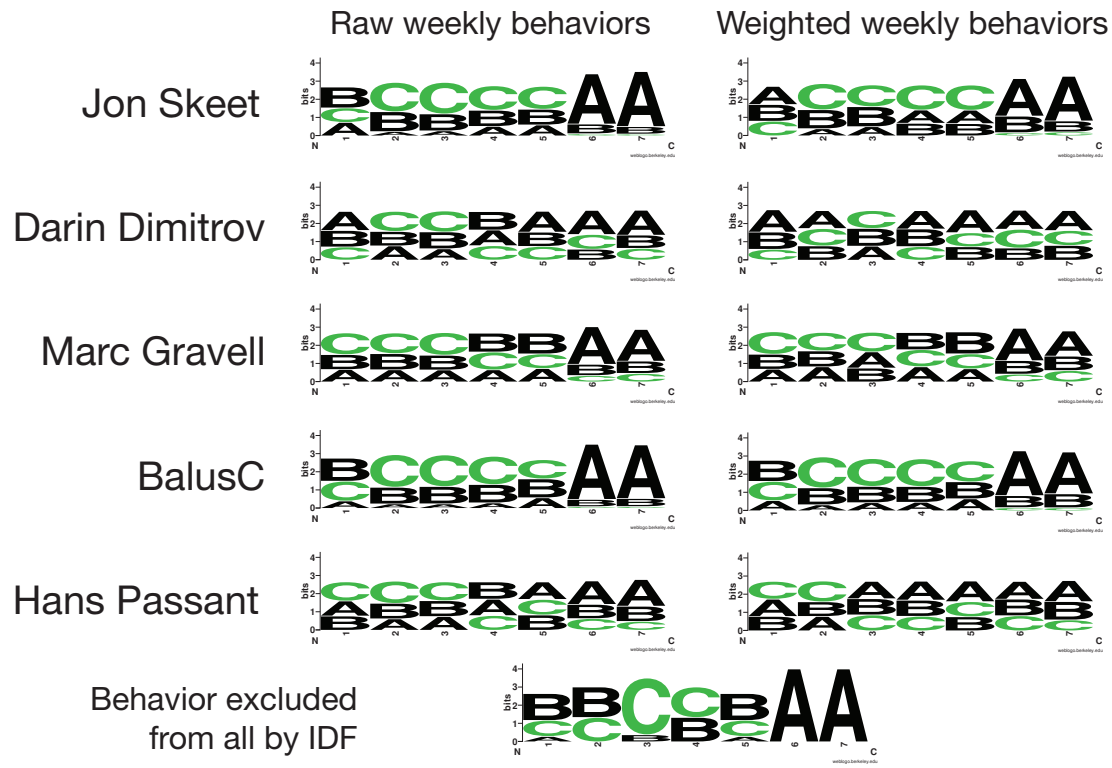


Figure 4.13: WebLogo visualization of weekly activity for top SO users. Here, letters (*A,B,C*) corresponds to (*low,medium,* and *high*) levels of activity. Note, that SAX-VSM pattern ranking process changed the effort distribution. The excluded behaviors are likely to represent a very common behavioral pattern: peaking at the mid-week performance and work-free weekends.

distribution by excluding common patterns and ranking.

The analysis of logo images for daily behaviors reveals that there are significant differences in the characteristic behavior patterns among the top SO users. For example, Jon Skeet's logo shows that his activity peaks in intervals (6AM - 9AM) and (3PM-9PM), which is confirmed by his public comment [212]: "*. . . I have a longish commute both ways each day: a 3G data dongle lets me answer questions during that time. I spend a fair amount of time in the evening on my computer for whatever reason (coding, writing talks or articles, etc) - I pop onto SO every so often. While at work, I tend to check SO while I have tests running, a deploy, or a build . . .*". Through personal communication I was also able to confirm the characteristic daily behavior of Marc Gravell, whose daily routines are structured by commute and other constraints, whereas Darin Dimitrov pointed out that his activity at SO are not structured in any way, which may explain that his logo images

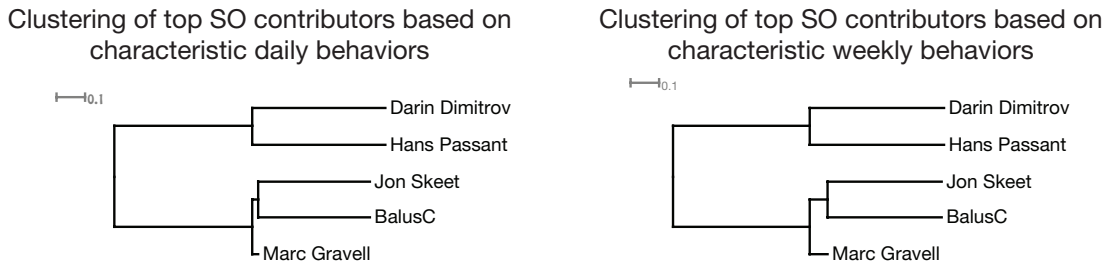


Figure 4.14: Clustering of StackOveflow user behaviors with STA. Note that D.Dimitrov's characteristic behaviors were found very different from those of J. Skeet, while their overall reputation scores are next to each other.

are more difficult to interpret (especially early morning (1AM-3AM) "C"s and considerably high weekend activity) and that the vector of his ranked behaviors was clustered separately of that with Skeet and Gravell as shown in Figure 4.14.

4.3.3.4 Discussion

While STA was able to discover user-characteristic patterns and WebLogo produced easily interpretable figures, clearly, these do not provide a sufficient knowledge why Jon Skeet's reputation is so high – in the daily behaviors study (3 green "C" at the top) and in the weekly behaviors study (4 green "C" at the top) his activity patterns were at the level with those of other users. It is hard to conclude it better than Jon's own comment [212]: "*. . . Often two answers may look quite similar, but one just about has an edge on the other - either it's explained just that bit better, or has one more piece of information, or a code sample. I'd like to hope that I have that sort of edge, and that that's why my answer would get more votes in that situation. But hey, I could easily be wrong! . . .*". Also, Skeet's habit of engaging into StackOverflow activities while en route is consistent with previously reported (non-scientific) observations concerning the impact of daily routines on productivity [213].

It was found that WebLogo provides a very efficient and reasonably effective way to convey the discretized trajectories summary, however, when a long time interval is considered it may fail to reveal the longitudinal phenomena evolution when compared with the rug plot-based visualization. Yet another WebLogo shortcoming is that while providing an excellent position-wise visualization, it fails to convey full pattern frequencies, which may affect the visualization effectiveness.

Note, that excluded by STA weighting daily behaviors correspond to a typical activity pattern

expected from an office worker [211], whose activity throughout the work week increases from 9AM, peaks at noon, and gradually degrades within the rest of the day. The excluded weekly behaviors also likely to be typical for office workers.

> The purpose of computing is insight, not numbers.
>
> Richard Hamming

CHAPTER 5
CONCLUSION

In this dissertation I have proposed the Software Trajectory Analysis – a generic framework for recurrent behaviors discovery from software process and product artifacts, whose ultimate premise is to provide means for empirical guidance of developers and project management in software development and decision-making processes. To aid the discovery of recurrent behaviors, I have also proposed a novel approach for time series classification, that not only enables the discovery and ranking of class-characteristic patterns, but, as I have shown, aids in interpretability of both: the classification results and the data specificity. This chapter summarizes my research, discusses its significance, and suggests future directions.

5.1 Dissertation summary

This dissertation covers a novel approach to the problem of recurrent behaviors discovery from software process artifacts. The research field-specific data type, that is *software trajectory*, its analysis paradigm, that is *Software Trajectory Analysis*, and a novel technique for time series classification and characteristic patterns discovery called *SAX-VSM* are proposed and evaluated.

In Chapter 1, I have described background for the explored research problem concerned with software process analysis. Specifically, I have emphasized the importance of an ability to discover recurrent behaviors offline by mining public software repositories. The concept of software trajectory, that is a temporally ordered sequence of software artifact measurements, and the Software Trajectory Analysis paradigm were introduced in the same Chapter.

Next, in Chapter 2, I have discussed software metrology and the relevant work from research area of mining software repositories, while focusing on the recurrent behaviors discovery.

In Chapter 3, addressing the problem of *unsupervised* knowledge discovery from software trajectories, and in particular the problem of time series class-characteristic patterns discovery, I have proposed and evaluated a novel technique for interpretable time series classification called SAX-VSM, which enables the discovery of class-characteristic patterns.

Finally, in Chapter 4, I have shown and evaluated the performance of a reference implementations of SAX-VSM-based Software Trajectory Analysis framework which provides end-to-end generic and customizable solution for the problem of recurrent behaviors discovery from software trajectories. The implemented system capabilities and limitations were also discussed.

5.2 Research summary

In contrast to the previous body of work in the area of software process analysis, that has been mostly concerned with identification of *previously known* behaviors for the purpose of software project management, the major distinction of this work is that it offers an ability to discover novel, *previously unknown* recurrent behaviors offline and in the automated manner.

5.3 Contributions

While the detailed list of contributions has been provided in Section 1.6, to summarize, I would like to emphasize two significant outcomes of my research.

First is the novel generic algorithm for interpretable time series classification which is yet to be used by the data mining community. Mining time series data will be an important area of research in coming years because of the growing ubiquity of time series. I expect SAX-VSM to play important role in the future development of time series data mining and to serve the practitioners with valuable insights.

The second important result of my research is that despite discovering best software trajectory class-characteristic patterns, their corresponding recurrent behaviors were found difficult to interpret without the domain knowledge and understanding of the studied phenomena context. This result emphasizes, that the software process design is inseparable from accounting for a project internal and external constraints as well as for human-specific aspects. This finding reflects the discussed in Section 1.4 specificities of OSS processes and shall aid in the future studies design.

5.4 Future work

A number of future directions suggests themselves. These can be divided into two categories - those that address current limitations of SAX-VSM and those that are concerned with the future STA-based research. Some immediate extension to the discussed in this dissertation work are:

- ***SAX-VSM ranking schema improvement.*** This addresses the possibility of a single software trajectory study, the two classes patterns ranking problem, and the patterns numerosity. Based on my current experience with the application of grammatical inference to discretized time series [196], I plan to develop a threshold-based extension of the SAX-VSM weighting schema, explore the possibility of a relevance-feedback algorithm application [194], and to implement a similar to the MDL principle [214] solution based on the minimal grammar size.

- ***Variable-length characteristic pattern discovery.*** This addresses the fixed sliding window length. It is possible that the best class-characteristic patterns have different lengths among classes, moreover, the capacity to work with variable length patterns should mitigate for the discussed in Section 4.1.1.3 effect of the class-characteristic pattern elimination by **idf**. Based on the previous application of grammatical inference to time series [215], and my own work [196], an extension of SAX-VSM was developed and currently being evaluated [216].

- ***Multivariate software trajectories mining.*** As I have pointed out in Section 4.1, it is highly desirable to extend STA capabilities to multivariate trajectories analysis. This direction was previously explored by Ordóñez et al in [217, 218] and the proposed solution can be used.

- ***In-depth study of a software project.*** This shall address the discovered recurrent behavior interpretation shortcoming and to allow a thorough evaluation of the proposed methodology through online interactions with the development team and project managers.

I expect this dissertation will continue to play important role in the future development of time series data mining and serve the practitioners in the field of software repository mining with valuable insights into this fascinating area of research.

BIBLIOGRAPHY

[1] D. R. Jones, C. D. Perttunen, and B. E. Stuckman. Lipschitzian optimization without the Lipschitz constant. *Journal of Optimization Theory and Applications*, 79(1):157–181, October 1993.

[2] M. Björkman and K. Holmström. Global optimization using the DIRECT algorithm in Matlab. In *in Matlab. Advanced Modeling and Optimization 1(2),17*, 1999.

[3] Byoung K. Yi and Christos Faloutsos. Fast time sequence indexing for arbitrary l_p norms. In *VLDB '00: Proceedings of the 26th International Conference on Very Large Data Bases*, pages 385–394, San Francisco, CA, USA, 2000. Morgan Kaufmann Publishers Inc.

[4] Jessica Lin, Eamonn Keogh, Li Wei, and Stefano Lonardi. Experiencing SAX: a novel symbolic representation of time series. *Data Mining and Knowledge Discovery*, 15(2):107–144, October 2007.

[5] Roger Pressman and Bruce Maxim. *Software Engineering: A Practitioner's Approach*. McGraw-Hill Science/Engineering/Math, 8 edition, January 2014.

[6] Kenneth S. Rubin. *Essential Scrum: A Practical Guide to the Most Popular Agile Process (Addison-Wesley Signature Series (Cohn))*. Addison-Wesley Professional, 1 edition, August 2012.

[7] G. Salton. *The SMART Retrieval System; Experiments in Automatic Document Processing*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1971.

[8] Kent Beck. *Test Driven Development: By Example*. Addison-Wesley Professional, 1 edition, November 2002.

[9] Kent Beck and Cynthia Andres. *Extreme Programming Explained: Embrace Change, 2nd Edition (The XP Series)*. Addison-Wesley, 2nd edition, November 2004.

[10] David T. Neal, Wendy Wood, Jennifer S. Labrecque, and Phillippa Lally. How do habits guide behavior? Perceived and actual triggers of habits in daily life. *Journal of Experimental Social Psychology*, 48(2):492–498, 2012.

[11] B. R. Andrews. Habit. *The American Journal of Psychology*, 14(2), 1903.

[12] R. N. Charette. Why software fails [software failure]. *Spectrum, IEEE*, 42(9):42–49, September 2005.

[13] Software engineering: Report of a conference sponsored by the NATO science committee, Garmisch, Germany, 7-11 Oct. 1968, Brussels, Scientific Affairs Division, NATO.

[14] Ian Sommerville. Software process models. *ACM Computing Surveys (CSUR)*, 28:269–271, March 1996.

[15] Watts S. Humphrey. *Managing the Software Process*. Addison-Wesley Professional, January 1989.

[16] ISO: Quality systems – model for quality assurance in design, development, production, installation and servicing. http://www.iso.org/iso/catalogue_detail.htm?csnumber=16534, 2000. Accessed: 2013-12-18.

[17] Watts S. Humphrey. Three process perspectives: Organizations, teams, and people. *Annals of Software Engineering*, 14(1):39–72, December 2002.

[18] Reidar Conradi. SPI frameworks: TQM, CMM, SPICE, ISO 9001, QIP experiences and trends - Norwegian SPIQ. In *Software Process Technology: 5th European Workshop, EWSPT '96, Nancy, France, October 9 - 11, 1996. Proceedings*, Lecture Notes in Artificial Intelligence. Springer, 1996.

[19] The Standish Group. CHAOS Report 2006. https://secure.standishgroup.com/reports/reports.php. Accessed: 2012-09-13.

[20] N. Wirth. A brief history of software engineering. *Annals of the History of Computing, IEEE*, 30(3):32–39, July 2008.

[21] Tom DeMarco. Software engineering: An idea whose time has come and gone? *Software, IEEE*, 26(4):96, July 2009.

[22] Kweku Ewusi-Mensah. *Software Development Failures*. The MIT Press, August 2003.

[23] Alistair Cockburn. *Agile Software Development*. Addison-Wesley Professional, October 2001.

[24] Walt Scacchi. Process models in software engineering. In *Encyclopedia of Software Engineering*. John Wiley & Sons, Inc., 2002.

[25] William A. Florac, Robert E. Park, and Anita D. Carleton. Practical software measurement: Measuring for process management and improvement. In *Software Engineering Measurement and Analysis*, pages 337–349, 1997.

[26] Robert Feldt, Lefteris Angelis, Richard Torkar, and Maria Samuelsson. Links between the personalities, views and attitudes of software engineers. *Information and Software Technology*, 52(6):611–624, June 2010.

[27] Tom DeMarco and Timothy Lister. *Peopleware: Productive Projects and Teams (Second Edition)*. Dorset House Publishing Company, Incorporated, 2nd edition, February 1999.

[28] S. T. Acuna, N. Juristo, and A. M. Moreno. Emphasizing human capabilities in software development. *Software, IEEE*, 23(2):94–101, 2006.

[29] E. Demirors, G. Sarmasik, and O. Demirors. The role of teamwork in software development: Microsoft case study. In *EUROMICRO 97. New Frontiers of Information Technology., Proceedings of the 23rd EUROMICRO Conference*, pages 129–133. IEEE, 1997.

[30] P. K. Janert. Software craftsmanship [book review]. *Software, IEEE*, 20(6):108–109, November 2003.

[31] Bill Pyritz. Craftsmanship versus engineering: computer programming – an art or a science? *Bell Labs Technical Journal*, 8(3):101–104, 2003.

[32] Robert English and Charles M. Schweik. Identifying success and tragedy of FLOSS commons: A preliminary classification of sourceforge.net projects. In *Proceedings of the First International Workshop on Emerging Trends in FLOSS Research and Development*, FLOSS '07, Washington, DC, USA, 2007. IEEE Computer Society.

[33] S. Richter. *Critique for the Open Source Development Model*. GRIN Verlag, 2007.

[34] Frederick P. Brooks. No silver bullet essence and accidents of software engineering. *Computer*, 20(4):10–19, April 1987.

[35] H. Goldstein. Who killed the virtual case file? [case management software]. *Spectrum, IEEE*, 42(9):24–35, September 2005.

[36] P. E. Ross. The exterminators [software bugs]. *Spectrum, IEEE*, 42(9):36–41, 2005.

[37] Winson W. Royce. Managing the development of a large software system. In *IEEE WESCON*, August 1970.

[38] Donald A. Norman. Interfacing thought: Cognitive aspects of human-computer interaction. chapter Cognitive Engineering – Cognitive Science, pages 325–336. MIT Press, Cambridge, MA, USA, 1987.

[39] Wil M. P. van der Aalst. *Process Mining: Discovery, Conformance and Enhancement of Business Processes*. Springer, 1st edition, April 2011.

[40] W. M. P. van der Aalst, A. Adriansyah, A. K. Alves de Medeiros, F. Arcieri, T. Baier, T. Blickle, R. P. Jagadeesh Chandra Bose, P. van den Brand, R. Brandtjen, J. C. A. M. Buijs, A. Burattin, J. Carmona, M. Castellanos, J. Claes, J. Cook, N. Costantini, F. Curbera, E. Damiani, M. de Leoni, P. Delias, B. F. van Dongen, M. Dumas, S. Dustdar, D. Fahland, D. R. Ferreira, W. Gaaloul, F. van Geffen, S. Goel, C. W. Gnther, A. Guzzo, P. Harmon, A. H. M. ter Hofstede, J. Hoogland, J. Espen Ingvaldsen, K. Kato, R. Kuhn, A. Kumar, M. La Rosa, F. Maggi, D. Malerba, R. S. Mans, A. Manuel, M. McCreesh, P. Mello, J. Mendling, M. Montali, H. Motahari Nezhad, M. zur Muehlen, J. Munoz-Gama, L. Pontieri, J. Ribeiro, A. Rozinat, H. Seguel Prez, R. Seguel Prez, M. Seplveda, J. Sinur, P. Soffer, M. S. Song, A. Sperduti,

G. Stilo, C. Stoel, K. Swenson, M. Talamo, W. Tan, C. Turner, J. Vanthienen, G. Varvaressos, H. M. W. Verbeek, M. Verdonk, R. Vigo, J. Wang, B. Weber, M. Weidlich, A. J. M. M. Weijters, L. Wen, M. Westergaard, and M. T. Wynn. Process mining manifesto. In *BPM 2011 Workshops, Part I*, volume 99, pages 169–194. Springer-Verlag, 2012.

[41] Hongbing Kou. *Automated Inference of Software Development Behaviors: Design, Implementation and Validation of Zorro for Test-Driven Development*. Ph.D. thesis, University of Hawaii, Department of Information and Computer Sciences, December 2007.

[42] Jonathan E. Cook and Alexander L. Wolf. Discovering models of software processes from event-based data. *ACM Trans. Softw. Eng. Methodol.*, 7(3):215–249, July 1998.

[43] Jonathan E. Cook. *Process discovery and validation through event-data analysis*. Ph.D. thesis, University of Colorado at Boulder, Boulder, CO, USA, 1996.

[44] Jonathan E. Cook, Zhidian Du, Chongbing Liu, and Alexander L. Wolf. Discovering models of behavior for concurrent workflows. *Computers in Industry - Special issue: Process/workflow mining*, 53(3):297–319, April 2004.

[45] Ming Huo, He Zhang, and Ross Jeffery. Detection of consistent patterns from process enactment data. In Qing Wang, Dietmar Pfahl, and David M. Raffo, editors, *Making Globally Distributed Software Development a Success Story*, volume 5007 of *Lecture Notes in Computer Science*, chapter 16, pages 173–185. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[46] Ming Huo, He Zhang, and Ross Jeffery. A systematic approach to process enactment analysis as input to software process improvement or tailoring. In *2006 13th Asia Pacific Software Engineering Conference (APSEC'06)*, pages 401–410. IEEE, December 2006.

[47] Steve McConnell. *Code Complete: A Practical Handbook of Software Construction, Second Edition*. Microsoft Press, 2nd edition, June 2004.

[48] Watts S. Humphrey. *A Discipline for Software Engineering*. Addison-Wesley Professional, 1st edition, January 1995.

[49] Watts S. Humphrey. *Managing Technical People: Innovation, Teamwork, and the Software Process*. Addison-Wesley Professional, 1 edition, November 1996.

[50] Dieter Rombach, Jürgen Münch, Alexis Ocampo, Watts S. Humphrey, and Dan Burton. Teaching disciplined software development. *Journal of Systems and Software*, 81(5):747–763, May 2008.

[51] Philip M. Johnson. Searching under the streetlight for useful software analytics. *IEEE Software*, July 2013.

[52] History of the OSI. http://opensource.org/history, 2006. Accessed: 2013-12-18.

[53] Coverity Scan Open Source Report, 2012. http://wpcme.coverity.com/wp-content/uploads/2012-Coverity-Scan-Report.pdf. Accessed: 2013-12-01.

[54] K. Crowston and B. Scozzi. Open source software projects as virtual organizations: competency rallying for software development. *Software, IEE Proceedings* -, 149(1):3–17, Feb 2002.

[55] Michael J. Gallivan. Striking a balance between trust and control in a virtual organization: a content analysis of open source software case studies. *Information Systems Journal*, 11(4):277–304, October 2001.

[56] Catharina Melian and Magnus Mähring. Lost and gained in translation: Adoption of open source software development at Hewlett-Packard. In Barbara Russo, Ernesto Damiani, Scott Hissam, Björn Lundell, and Giancarlo Succi, editors, *Open Source Development, Communities and Quality*, volume 275 of *IFIP The International Federation for Information Processing*, pages 93–104. Springer US, 2008.

[57] G. Gaughan, B. Fitzgerald, and M. Shaikh. An examination of the use of open source software processes as a global software development solution for commercial software engineering. In *Software Engineering and Advanced Applications, 2009. SEAA '09. 35th Euromicro Conference on*, pages 20–27. IEEE, August 2009.

[58] Chris Jensen and Walt Scacchi. Simulating an automated approach to discovery and modeling of open source software development processes. In *In Proceedings of Software Process Simulation and Modeling Workshop*, 2003.

[59] Chris Jensen and Walt Scacchi. Guiding the discovery of open source software processes with a reference model. In Joseph Feller, Brian Fitzgerald, Walt Scacchi, and Alberto Sillitti, editors, *Open Source Development, Adoption and Innovation*, volume 234 of *IFIP – The International Federation for Information Processing*, pages 265–270. Springer US, 2007.

[60] Chris Jensen and Walt Scacchi. Process modeling across the web information infrastructure. *Software Process: Improvement and Practice*, 10(3):255–272, 2005.

[61] Abram Hindle, Michael W. Godfrey, and Richard C. Holt. Release pattern discovery via partitioning: Methodology and case study. In *Proceedings of the 29th International Conference on Software Engineering Workshops*, Washington, DC, USA, 2007. IEEE Computer Society.

[62] W. Scacchi. Understanding the requirements for developing open source software systems. *Software, IEE Proceedings* -, 149(1):24–39, February 2002.

[63] MSR 2004, international workshop on mining software repositories. In *Software Engineering, 2004. ICSE 2004. Proceedings. 26th International Conference on*, pages 770–771, May 2004.

[64] Ahmed E. Hassan. The road ahead for mining software repositories. In *2008 Frontiers of Software Maintenance*, pages 48–57. IEEE, September 2008.

[65] Tim Menzies, Bora Caglayan, Ekrem Kocaguneli, Joe Krall, Fayola Peters, and Burak Turhan. The PROMISE repository of empirical software engineering data. http://promisedata.googlecode.com, June 2012.

[66] A. Hindle, M. W. Godfrey, and R. C. Holt. Mining recurrent activities: Fourier analysis of change events. In *Software Engineering - Companion Volume, 2009. ICSE-Companion 2009. 31st International Conference on*, pages 295–298. IEEE, May 2009.

[67] W. Vanderaalst, B. Vandongen, J. Herbst, L. Maruster, G. Schimm, and A. Weijters. Workflow mining: A survey of issues and approaches. *Data & Knowledge Engineering*, 47(2):237–267, November 2003.

[68] Jana Samalikova, Rob Kusters, Jos Trienekens, Ton Weijters, and Paul Siemons. Toward objective software process information: Experiences from a case study. *Software Quality Control*, 19(1):101–120, March 2011.

[69] Giuliano Antoniol, Vincenzo F. Rollo, and Gabriele Venturi. Linear predictive coding and cepstrum coefficients for mining time variant information from software repositories. In *Proceedings of the 2005 international workshop on Mining software repositories*, volume 30 of *MSR '05*, pages 1–5, New York, NY, USA, 2005. ACM.

[70] Marsha Pomeroy-Huff, Julia Mullaney, Robert Cannon, and Mark Seburn. The personal software process (PSP) body of knowledge, version 1.0. Technical report, Software Engineering Institute, Pittsburgh, PA 15213, 2008.

[71] Watts S. Humphrey. *A Discipline for Software Engineering*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[72] ISO: Information technology – process assessment. http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=38932, 2008. Accessed: 2013-12-18.

[73] Albert Endres and Dieter Rombach. *A Handbook of Software and Systems Engineering: Empirical Observations, Laws and Theories*. Addison-Wesley, Illustrated edition, May 2003.

[74] P. M. Johnson, Hongbing Kou, M. Paulding, Qin Zhang, A. Kagawa, and T. Yamashita. Improving software development management through software project telemetry. *Software, IEEE*, 22(4):76–85, July 2005.

[75] P. M. Johnson. Requirement and design trade-offs in Hackystat: an in-process software engineering measurement and analysis system. In *Empirical Software Engineering and Measurement, 2007. ESEM 2007. First International Symposium on*, pages 81–90. IEEE, 2007.

[76] Hongbing Kou, Philip M. Johnson, and Hakan Erdogmus. Operational definition and automated inference of test-driven development with Zorro. *Automated Software Engineering*, 17(1):57–85, 2010.

[77] P.M. Johnson and Hongbing Kou. Automated recognition of Test-Driven Development with Zorro. In *Agile Conference (AGILE), 2007*, pages 15–25, Aug 2007.

[78] Hui Ding, Goce Trajcevski, Peter Scheuermann, Xiaoyue Wang, and Eamonn Keogh. Querying and mining of time series data: Experimental comparison of representations and distance measures. *Proc. VLDB Endow.*, 1(2):1542–1552, August 2008.

[79] Romain Briandet, E. Katherine Kemsley, and Reginald H. Wilson. Discrimination of arabica and robusta in instant coffee by fourier transform infrared spectroscopy and chemometrics. *Journal of agricultural and food chemistry*, 44(1):170–174, Jan 1996.

[80] Eamonn Keogh, Li Wei, Xiaopeng Xi, Sang-hee Lee, and Michail Vlachos. LB_Keogh supports exact indexing of shapes under rotation invariance with arbitrary representations and distance measures. In *VLDB, 2006*, pages 882–893, 2006.

[81] Xiaoyue Wang, Lexiang Ye, Eamonn Keogh, and Christian Shelton. Annotating historical archives of images. In *Proceedings of the 8th ACM/IEEE-CS joint conference on Digital libraries*, JCDL '08, pages 341–350, New York, NY, USA, 2008. ACM.

[82] Frank Höppner. Discovery of temporal patterns. Learning rules about the qualitative behaviour of time series. In *Proceedings of the 5th European Conference on Principles of Data Mining and Knowledge Discovery*, PKDD '01, pages 192–203, London, UK, UK, 2001. Springer-Verlag.

[83] Jiawei Han, Guozhu Dong, and Yiwen Yin. Efficient mining of partial periodic patterns in time series database. In *Data Engineering, 1999. Proceedings., 15th International Conference on*, pages 106–115. IEEE, March 1999.

[84] Gautam Das, King-Ip Lin, Heikki Mannila, Gopal Renganathan, and Padhraic Smyth. Rule discovery from time series. *KDD*, 98:16–22, 1998.

[85] Eamonn Keogh and M. Pazzani. An enhanced representation of time series which allows fast and accurate classification, clustering and relevance feedback. In R. Agrawal, P. Stolorz, and G. Piatetsky-Shapiro, editors, *Fourth International Conference on Knowledge Discovery and Data Mining (KDD'98)*, pages 239–241, New York City, NY, 1998. ACM Press.

[86] Fabian Mörchen and Alfred Ultsch. Efficient mining of understandable patterns from multivariate interval time series. *Data Mining and Knowledge Discovery*, 15(2):181–215, October 2007.

[87] Bill Chiu, Eamonn Keogh, and Stefano Lonardi. Probabilistic discovery of time series motifs. In *Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD '03, pages 493–498, New York, NY, USA, 2003. ACM.

[88] Lexiang Ye and Eamonn Keogh. Time series shapelets: a novel technique that allows accurate, interpretable and fast classification. *Data Mining and Knowledge Discovery*, 22(1-2):149–182, January 2011.

[89] Abdullah Mueen, Eamonn Keogh, and Neal Young. Logical-shapelets: an expressive primitive for time series classification. In *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD '11, pages 1154–1162, New York, NY, USA, 2011. ACM.

[90] Jesin Zakaria, Abdullah Mueen, and Eamonn Keogh. Clustering time series using Unsupervised-Shapelets. In *Proceedings of the 2012 IEEE 12th International Conference on Data Mining*, ICDM '12, pages 785–794, Washington, DC, USA, 2012. IEEE Computer Society.

[91] Jessica Lin, Rohan Khade, and Yuan Li. Rotation-invariant similarity in time series using Bag-Of-Patterns representation. *Journal of Intelligent Information Systems*, 39(2):287–315, October 2012.

[92] P. Patel, E. Keogh, J. Lin, and S. Lonardi. Mining motifs in massive time series databases. In *Data Mining, 2002. ICDM 2003. Proceedings. 2002 IEEE International Conference on*, pages 370–377. IEEE, 2002.

[93] Pavel Senin. Software trajectory analysis: An empirically based method for automated software process discovery. In *Proceedings of the Fifth International Doctoral Symposium on Empirical Software Engineering*, Bolzano-Bozen, Italy, September 2010.

[94] Pavel Senin and Sergey Malinchik. SAX-VSM: Interpretable time series classification using SAX and vector space model. In *Data Mining (ICDM), 2013 IEEE 13th International Conference on*, pages 1175–1180. IEEE, December 2013.

[95] R. A. DeMillo, R. J. Lipton, Georgia I. Information, and Science. *Software Project Forecasting*. Defense Technical Information Center, 1980.

[96] Pavel Senin. Recognizing recurrent development behaviors corresponding to Android OS release life-cycle. In *Software Engineering Research and Practice*, May 2012.

[97] Pavel Senin. JMotif project homepage. reference implementation of SAX-VSM algorithm. https://code.google.com/p/jmotif/.

[98] Pavel Senin. Software Trajectory Analysis project homepage. reference implementation of STA framework. https://code.google.com/p/hackystat-ui-trajectory/.

[99] Hadi Hemmati, Sarah Nadi, Olga Baysal, Oleksii Kononenko, Wei Wang, Reid Holmes, and Michael W. Godfrey. The MSR cookbook: mining a decade of research. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, MSR '13, pages 343–352, Piscataway, NJ, USA, 2013. IEEE Press.

[100] Daniel M. German. Mining CVS repositories, the softchange experience. In *Proceedings of the First International Workshop on Mining Software Repositories*, pages 17–21, Edinburg, Scotland, UK, 2004.

[101] Gregorio Robles. *Empirical Software Engineering Research on Libre Software: Data Sources, Methodologies and Results*. Ph.D. thesis, Universidad Rey Juan Carlos, February 2006.

[102] Abdullah Mueen. Time series motif discovery: dimensions and applications. *WIREs Data Mining Knowl Discov*, 4(2):152–159, March 2014.

[103] Thomas Zimmermann, Nachiappan Nagappan, Harald Gall, Emanuel Giger, and Brendan Murphy. Cross-project defect prediction: A large scale experiment on data vs. domain vs. process. In *Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC/FSE '09, pages 91–100, New York, NY, USA, 2009. ACM.

[104] Norman E. Fenton and Martin Neil. Software metrics: Roadmap. In *Proceedings of the Conference on The Future of Software Engineering*, ICSE '00, pages 357–370, New York, NY, USA, 2000. ACM.

[105] Tom Gilb. *Software metrics*. Winthrop Publishers, 1977.

[106] G. Redig and M. Swanson. Total quality management for software development. In *Computer-Based Medical Systems, 1993. Proceedings of Sixth Annual IEEE Symposium on*, pages 301–306. IEEE, June 1993.

[107] W. S. Humphrey. Using a defined and measured personal software process. *Software, IEEE*, 13(3):77–88, May 1996.

[108] Huzefa Kagdi, Michael L. Collard, and Jonathan I. Maletic. A survey and taxonomy of approaches for mining software repositories in the context of software evolution. *Journal of Software Maintenance and Evolution: Research and Practice*, 19(2):77–131, March 2007.

[109] Huzefa Kagdi, Michael L. Collard, and Jonathan I. Maletic. Towards a taxonomy of approaches for mining of source code repositories. *SIGSOFT Softw. Eng. Notes*, 30(4):1–5, May 2005.

[110] Ned Chapin. A measure of software complexity. In *AFIPS National Computer Conference*, pages 995–1002, 1979.

[111] N. Fenton. Software measurement: a necessary scientific basis. *Software Engineering, IEEE Transactions on*, 20(3):199–206, March 1994.

[112] M. M. Lehman. Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE*, 68(9):1060–1076, 1980.

[113] E. Shihab, Y. Kamei, and P. Bhattacharya. Mining challenge 2012: The Android platform. In *Mining Software Repositories (MSR), 2012 9th IEEE Working Conference on*, pages 112–115. IEEE, June 2012.

[114] Alberto Bacchelli. Mining challenge 2013: Stack Overflow. In *The 10th Working Conference on Mining Software Repositories (MSR)*, 2013.

[115] Tim Menzies. Guest editorial for the special section on BEST PAPERS from the 2011 conference on predictive models in software engineering (PROMISE). *Information & Software Technology*, 55(8):1477–1478, 2013.

[116] M. D'Ambros, M. Lanza, and R. Robbes. An extensive comparison of bug prediction approaches. In *Mining Software Repositories (MSR), 2010 7th IEEE Working Conference on*, pages 31–41. IEEE, May 2010.

[117] Les Gasser, Gabriel Ripoche, and Robert J. Sandusky. Research infrastructure for empirical science of F/OSS. In *Proc. Intern. Workshop on Mining Software Repositories*, 2004.

[118] H. Kagdi, J. I. Maletic, and B. Sharif. Mining software repositories for traceability links. In *Program Comprehension, 2007. ICPC '07. 15th IEEE International Conference on*, pages 145–154. IEEE, June 2007.

[119] A. Begel, Yit P. Khoo, and T. Zimmermann. Codebook: discovering and exploiting relationships in software repositories. In *Software Engineering, 2010 ACM/IEEE 32nd International*

Conference on, volume 1 of *ICSE '10*, pages 125–134, New York, NY, USA, May 2010. IEEE.

[120] Jim Buckley, Tom Mens, Matthias Zenger, Awais Rashid, and Günter Kniesel. Towards a taxonomy of software change: Research articles. *Journal of Software Maintenance and Evolution: Research and Practice*, 17(5):309–332, September 2005.

[121] Romain Robbes. Mining a Change-Based software repository. In *Proceedings of the Fourth International Workshop on Mining Software Repositories*, MSR '07, Washington, DC, USA, 2007. IEEE Computer Society.

[122] Lile Hattori and Michele Lanza. Mining the history of synchronous changes to refine code ownership. In *Proceedings of the 2009 6th IEEE International Working Conference on Mining Software Repositories*, volume 0 of *MSR '09*, pages 141–150, Washington, DC, USA, 2009. IEEE Computer Society.

[123] Michael W. Godfrey, Ahmed E. Hassan, James Herbsleb, Gail C. Murphy, Martin Robillard, Prem Devanbu, Audris Mockus, Dewayne E. Perry, and David Notkin. Future of mining software archives: a roundtable. *Software, IEEE*, 26(1):67–70, January 2009.

[124] Bart Massey. Longitudinal analysis of long-timescale open source repository data. In *Proceedings of the 2005 Workshop on Predictor Models in Software Engineering*, PROMISE '05, pages 1–5, New York, NY, USA, 2005. ACM.

[125] Benjamin Livshits and Thomas Zimmermann. DynaMine: Finding common error patterns by mining software revision histories. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, volume 30 of *ESEC/FSE-13*, pages 296–305, New York, NY, USA, September 2005. ACM.

[126] Thomas J. Ostrand and Elaine J. Weyuker. A tool for mining defect-tracking systems to predict fault-prone files. In *1st ICSE workshop on mining software repositories: Proceedings of the 1st ICSE Workshop on Mining Software Repositories*. IET, 2004.

[127] Annie T. T. Ying, James L. Wright, and Steven Abrams. Source code that talks: An exploration of Eclipse task comments and their implication to repository mining. In *Proceedings of the 2005 International Workshop on Mining Software Repositories*, MSR '05, pages 1–5, New York, NY, USA, 2005. ACM.

[128] Shih K. Huang and Kang M. Liu. Mining version histories to verify the learning process of legitimate peripheral participants. *SIGSOFT Softw. Eng. Notes*, 30(4):1–5, May 2005.

[129] Kartik Bajaj, Karthik Pattabiraman, and Ali Mesbah. Mining questions asked by web developers. In *Proceedings of the Working Conference on Mining Software Repositories (MSR)*. ACM, 2014.

[130] Rahul Venkataramani, Atul Gupta, Allahbaksh M. Asadullah, Basavaraju Muddu, and Vasudev D. Bhat. Discovery of technical expertise from open source code repositories. In *WWW (Companion Volume)*, pages 97–98, 2013.

[131] Joshua Saxe, David Mentis, and Christopher Greamo. Mining web technical discussions to identify malware capabilities. In *ICDCS Workshops*, pages 1–5, 2013.

[132] David Kavaler, Daryl Posnett, Clint Gibler, Hao Chen, Premkumar T. Devanbu, and Vladimir Filkov. Using and asking: APIs used in the android market and asked about in StackOverflow. In *SocInfo*, pages 405–418, 2013.

[133] Mario Linares-Vásquez, Bogdan Dit, and Denys Poshyvanyk. An exploratory analysis of mobile development issues using Stack Overflow. In *Proceedings of the 10th International Working Conference on Mining Software Repositories*, pages 93–96. IEEE, 2013.

[134] Joshua Charles Campbell, Chenlei Zhang, Zhen Xu, Abram Hindle, and James Miller. Deficient documentation detection: A methodology to locate deficient project documentation using topic analysis. In *Proceedings of the 10th International Working Conference on Mining Software Repositories*, pages 57–60. IEEE, 2013.

[135] Yla R. Tausczik, Aniket Kittur, and Robert E. Kraut. Collaborative problem solving: a study of MathOverflow. In *Proceedings of the 17th ACM Conference on Computer Supported*

Cooperative Work and Social Computing, CSCW '14, pages 355–367, New York, NY, USA, 2014. ACM.

[136] Bogdan Vasilescu, Alexander Serebrenik, Premkumar T. Devanbu, and Vladimir Filkov. How social Q&A sites are changing knowledge sharing in open source software communities. In *Proceedings of the 17th ACM Conference on Computer Supported Cooperative Work and Social Computing*, pages 342–354. ACM, 2014.

[137] Dennis Schenk and Mircea Lungu. Geo-locating the knowledge transfer in StackOverflow. In *Proceedings of the 2013 International Workshop on Social Software Engineering*, pages 21–24. ACM, 2013.

[138] Amiangshu Bosu, Christopher S Corley, Dustin Heaton, Debarshi Chatterji, Jeffrey C Carver, and Nicholas A Kraft. Building reputation in StackOverflow: An empirical investigation. In *Proceedings of the 10th International Working Conference on Mining Software Repositories*, pages 89–92. IEEE, 2013.

[139] Alexandru-Lucian Ginsca and Adrian Popescu. User profiling for answer quality assessment in Q&A communities. In *DUBMOD@CIKM*, pages 25–28, 2013.

[140] Sunghun Kim, Thomas Zimmermann, Miryung Kim, Ahmed Hassan, Audris Mockus, Tudor Girba, Martin Pinzger, E. James Whitehead, and Andreas Zeller. TA-RE: An exchange language for mining software repositories. In *Proceedings of the 2006 International Workshop on Mining Software Repositories*, MSR '06, pages 22–25, New York, NY, USA, 2006. ACM.

[141] Rui Ding, Qiang Fu, Jian G. Lou, Qingwei Lin, Dongmei Zhang, Jiajun Shen, and Tao Xie. Healing online service systems via mining historical issue repositories. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ASE 2012, pages 318–321, New York, NY, USA, 2012. ACM.

[142] Dongmei Zhang, Yingnong Dang, Jian G. Lou, Shi Han, Haidong Zhang, and Tao Xie. Software analytics as a learning case in practice: Approaches and experiences. In *Proceedings*

of the International Workshop on Machine Learning Technologies in Software Engineering, MALETS '11, pages 55–58, New York, NY, USA, 2011. ACM.

[143] T. Zimmermann, A. Zeller, P. Weissgerber, and S. Diehl. Mining version histories to guide software changes. *Software Engineering, IEEE Transactions on*, 31(6):429–445, June 2005.

[144] A. Mockus and L. G. Votta. Identifying reasons for software changes using historic databases. In *Software Maintenance, 2000. Proceedings. International Conference on*, volume 0, pages 120–130, Los Alamitos, CA, USA, August 2000. IEEE.

[145] David Atkins, Thomas Ball, Todd Graves, and Audris Mockus. Using version control data to evaluate the impact of software tools: A case study of the version editor. In *IEEE Transactions on Software Engineering*, pages 324–333, 2002.

[146] Bonsai project. https://wiki.mozilla.org/Bonsai, 2014. Accessed: 2014-04-02.

[147] Kenny Wong, Warren Blanchet, Ying Liu, Curtis Schofield, Eleni Stroulia, and Zhenchang Xing. JRefleX: Towards supporting small student software teams. In *Proceedings of the 2003 OOPSLA Workshop on Eclipse Technology eXchange*, eclipse '03, pages 50–54, New York, NY, USA, 2003. ACM.

[148] Gregorio Robles, Stefan Koch, Jesús M. González-Barahona, and Juan Carlos. Remote analysis and measurement of libre software systems by means of the CVSAnalY tool. In *Proceedings of the 2nd ICSE Workshop on Remote Analysis and Measurement of Software Systems (RAMSS)*, pages 51–55, 2004.

[149] Daniel German. Automating the measurement of open source projects. In *In Proceedings of the 3rd Workshop on Open Source Software Engineering*, pages 63–67, 2003.

[150] Vladimir Rubin, Christian W. Günther, Wil M. P. Aalst, Ekkart Kindler, Boudewijn F. Dongen, and Wilhelm Schäfer. *Process Mining Framework for Software Processes*, volume 4470 of *Lecture Notes in Computer Science*, chapter 15, pages 169–181. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007.

[151] Huzefa Kagdi, Shehnaaz Yusuf, and Jonathan I. Maletic. Mining sequences of changed-files from version histories. In *Proceedings of the 2006 international workshop on Mining software repositories*, MSR '06, pages 47–53, New York, NY, USA, 2006. ACM.

[152] I. Herraiz, J. M. Gonzalez-Barahona, and G. Robles. Forecasting the number of changes in Eclipse using time series analysis. In *Mining Software Repositories, 2007. ICSE Workshops MSR '07. Fourth International Workshop on*, page 32, May 2007.

[153] Harvey Siy, Parvathi Chundi, and Mahadevan Subramaniam. Summarizing developer work history using time series segmentation: Challenge report. In *Proceedings of the 2008 International Working Conference on Mining Software Repositories*, MSR '08, pages 137–140, New York, NY, USA, 2008. ACM.

[154] Edward R. Tufte. *The Visual Display of Quantitative Information*. Graphics Press USA, 2001.

[155] Xiaoyue Wang, Abdullah Mueen, Hui Ding, Goce Trajcevski, Peter Scheuermann, and Eamonn Keogh. Experimental comparison of representation methods and distance measures for time series data. *Data Mining and Knowledge Discovery*, 26(2):275–309, February 2013.

[156] Eamonn Keogh and Shruti Kasetty. On the need for time series data mining benchmarks: A survey and empirical demonstration. *Data Min. Knowl. Discov.*, 7(4):349–371, October 2003.

[157] Steven L. Salzberg. On comparing classifiers: Pitfalls to avoid and a recommended approach. In *Data Mining and Knowledge Discovery*, volume 1, pages 317–328. Kluwer Academic Publishers, 1997.

[158] Xiaopeng Xi, Eamonn Keogh, Christian Shelton, Li Wei, and Chotirat A. Ratanamahatana. Fast time series classification using numerosity reduction. In *Proceedings of the 23rd international conference on Machine learning*, ICML '06, pages 1033–1040, New York, NY, USA, 2006. ACM.

[159] Pavel Senin. Dynamic time warping algorithm review. *University Of Hawaii, ICS CSDL Technical report*, 2008.

[160] Rakesh Agrawal, Christos Faloutsos, and Arun N. Swami. Efficient similarity search in sequence databases. In *Proceedings of the 4th International Conference on Foundations of Data Organization and Algorithms*, FODO '93, pages 69–84, London, UK, UK, 1993. Springer-Verlag.

[161] Jason Lines, Luke M. Davis, Jon Hills, and Anthony Bagnall. A shapelet transform for time series classification. In *Proceedings of the 18th ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD '12, pages 289–297, New York, NY, USA, 2012. ACM.

[162] T. Rakthanamanon and E. Keogh. Fast-Shapelets: A scalable algorithm for discovering time series shapelets. In *Proceedings of the SIAM Intl. Conf. on Data Mining*, 2013.

[163] Eamonn Keogh, Jessica Lin, and Ada Fu. HOT SAX: Efficiently finding the most unusual time series subsequence. In *Proceedings of the Fifth IEEE International Conference on Data Mining*, ICDM '05, pages 226–233, Washington, DC, USA, 2005. IEEE Computer Society.

[164] Sotiris Kotsiantis and Dimitris Kanellopoulos. Discretization techniques: A recent survey. In *GESTS International Transactions on Computer Science and Engineering*, volume 1, pages 47–58. 2006.

[165] Dina Goldin and Paris Kanellakis. On similarity queries for time-series data: Constraint specification and implementation. In *Principles and Practice of Constraint Programming – CP '95*, pages 137–153. 1995.

[166] Eamonn Keogh, Kaushik Chakrabarti, Michael Pazzani, and Sharad Mehrotra. Dimensionality reduction for fast similarity search in large time series databases. *Knowledge and Information Systems*, 3(3):263–286, August 2001.

[167] Richard J. Larsen and Morris L. Marx. *Introduction to Mathematical Statistics and Its Applications (5th Edition)*. Pearson, 5 edition, January 2011.

[168] Hui Ding, Goce Trajcevski, Peter Scheuermann, Xiaoyue Wang, and Eamonn Keogh. Querying and mining of time series data: experimental comparison of representations and distance measures. *Proc. VLDB Endow.*, 1(2):1542–1552, August 2008.

[169] Eamonn Keogh, Jessica Lin, and Wagner Truppel. Clustering of time series subsequences is meaningless: Implications for previous and future research. In *Proceedings of the Third IEEE International Conference on Data Mining*, ICDM '03, Washington, DC, USA, 2003. IEEE Computer Society.

[170] R. W. Hamming. Error detecting and error correcting codes. *Bell System Techincal Journal*, 29:147–160, 1950.

[171] Gerard Salton, A. Wong, and C. S. Yang. A vector space model for automatic indexing. Technical report, Ithaca, NY, USA, 1974.

[172] M. G. Baydogan, G. Runger, and E. Tuv. A Bag-of-Features framework to classify time series. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 35(11):2796–2802, November 2013.

[173] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to Information Retrieval*. Cambridge University Press, New York, NY, USA, 2008.

[174] E. Keogh, Q. Zhu, B. Hu, Y. Hao, X. Xi, L. Wei, and C. Ratanamahatana. The UCR time series classification/clustering homepage. http://www.cs.ucr.edu/~eamonn/time_series_data/. Accessed: 2013-12-01.

[175] Osama Al-Jowder, E. K. Kemsley, and Reginald H. Wilson. Detection of adulteration in cooked meat products by mid-infrared spectroscopy. *Journal of agricultural and food chemistry*, 50(6):1325–1329, March 2002.

[176] Saito Naoki. *Local feature extraction and its application using a library of bases*. Ph.D. thesis, Yale University, 1994.

[177] Geurts Pierre. *Contributions to decision tree induction: bias/variance tradeoff and time series classification*. Ph.D. thesis, University of Lige, Belgium, 2002.

[178] Chotirat Ann Ratanamahatana and Eamonn Keogh. Making time-series classification more accurate using learned constraints. In *In proc. of SDM Intl Conf*, pages 11–22, 2004.

[179] Michael A. Dirr. *Manual of Woody Landscape Plants: Their Identification, Ornamental Characteristics, Culture, Propogation and Uses*. Stipes Pub Llc, 6, revised edition.

[180] A. Gandhi. Content-based image retrieval: plant species identification. Master's thesis, Oregon State University, 2002.

[181] Richard O. Duda, Peter E. Hart, and David G. Stork. *Pattern Classification (Pt.1)*. Wiley-Interscience, 2 edition, November 2000.

[182] Stephen C. Johnson. Hierarchical clustering schemes. *Psychometrika*, 32(3):241–254, September 1967.

[183] J. B. MacQueen. Some methods for classification and analysis of MultiVariate observations. In Le M. L. Cam and J. Neyman, editors, *Proc. of the fifth Berkeley Symposium on Mathematical Statistics and Probability*, volume 1, pages 281–297. University of California Press, 1967.

[184] P. S. Bradley, Usama Fayyad, and Cory Reina. Scaling clustering algorithms to large databases. In *Knowledge Discovery and Data Mining*, pages 9–15, 1998.

[185] Ying Zhao and George Karypis. Empirical and theoretical comparisons of selected criterion functions for document clustering. *Machine Learning*, 55(3):311–331, June 2004.

[186] Inderjit S. Dhillon and Dharmendra S. Modha. Concept decompositions for large sparse text data using clustering. *Machine Learning*, 42(1-2):143–175, January 2001.

[187] A. L. Goldberger, L. A. Amaral, L. Glass, J. M. Hausdorff, P. C. Ivanov, R. G. Mark, J. E. Mietus, G. B. Moody, C. K. Peng, and H. E. Stanley. PhysioBank, PhysioToolkit, and Phys-

ioNet: components of a new research resource for complex physiologic signals. *Circulation*, 101(23):e215–e220, June 2000.

[188] R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2014.

[189] J. A. Hartigan and M. A. Wong. Algorithm AS 136: A k-means clustering algorithm. *Applied Statistics*, 28(1):100–108, 1979.

[190] Martin Gavrilov, Dragomir Anguelov, Piotr Indyk, and Rajeev Motwani. Mining the stock market (extended abstract): which measure is best? In *Proceedings of the sixth ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD '00, pages 487–496, New York, NY, USA, 2000. ACM.

[191] Pavel Senin. Software trajectory analysis: an empirically based method for automated software process discovery. Dissertation proposal. University of Hawai'i at Manōa, 2009.

[192] jGit project: Java implementation of the Git version control system. http://www.eclipse.org/jgit/. Accessed: 2014-10-17.

[193] Siva Prasad Reddy. *Java Persistence with MyBatis 3*. Packt Publishing, June 2013.

[194] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to Information Retrieval*. Cambridge University Press, New York, NY, USA, 2008.

[195] Michael Widenius and Davis Axmark. *MySQL Reference Manual*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1st edition, 2002.

[196] P. Senin, J. Lin, X. Wang, T. Oates, S. Gandhi, A. P. Boedihardjo, C. Chen, S. Frankenstein, and M. Lerner. GrammarViz 2.0: a tool for grammar-based pattern discovery in time series. In T. Calders, editor, *ECML/PKDD 2014*, number LNCS 8726, pages 468–472.

[197] Diagnosis of multiple cancer types by shrunken centroids of gene expression. *Proceedings of the National Academy of Sciences*, 99(10):6567–6572, May 2002.

[198] R. Agrawal and R. Srikant. Mining sequential patterns. In *Data Engineering, 1995. Proceedings of the Eleventh International Conference on*, volume 0, pages 3–14, Los Alamitos, CA, USA, March 1995. IEEE.

[199] W. W. Royce. Managing the development of large software systems: Concepts and techniques. In *Proceedings of the 9th International Conference on Software Engineering*, ICSE '87, pages 328–338, Los Alamitos, CA, USA, 1987. IEEE Computer Society Press.

[200] J. Boegh. A new standard for quality requirements. *Software, IEEE*, 25(2):57–63, March 2008.

[201] Ivar Jacobson, Grady Booch, and James Rumbaugh. *The Unified Software Development Process (Paperback) (Addison-Wesley Object Technology Series)*. Addison-Wesley Professional, 1 edition, February 1999.

[202] Ian Sommerville. *Software engineering*. Pearson, 9th edition, March 2011.

[203] Android API levels. http://en.wikipedia.org/wiki/Android_version_history.

[204] Barry W. Boehm. *Software Engineering Economics*. Prentice Hall, 1 edition, November 1981.

[205] C. E. Walston and C. P. Felix. A method of programming measurement and estimation. *IBM Systems Journal*, 16(1):54–73, March 1977.

[206] Capers Jones. *Software Assessments, Benchmarks, and Best Practices*. Addison-Wesley Professional, 1 edition, May 2000.

[207] PostgreSQL, Contributor Profiles. http://www.postgresql.org/community/contributors/. Accessed: 2013-12-18.

[208] PostgreSQL, Commit Fest documentation. https://commitfest.postgresql.org/. Accessed: 2014-04-02.

[209] Seymour Geisser. *Predictive Inference (Chapman & Hall/CRC Monographs on Statistics & Applied Probability)*. Chapman and Hall/CRC, softcover reprint of the original 1st ed. 1993 edition, June 1993.

[210] Gavin E. Crooks, Gary Hon, John-Marc M. Chandonia, and Steven E. Brenner. WebLogo: a sequence logo generator. *Genome research*, 14(6):1188–1190, June 2004.

[211] Marta C. Gonzalez Jiang, Shan and Joseph Ferreira. Understanding the link between urban activity destinations and human travel patterns. In *Proceedings of the 12th International Conference on Computers in Urban Planning & Urban Management, CUPUM 2011*, 2011.

[212] Top users on StackOverflow: slackers or superstars? http://meta.stackexchange.com/questions/12468/top-users-on-stackoverflow-slackers-or-superstars. Accessed: 2014-10-17.

[213] Twyla Tharp. *The Creative Habit: Learn It and Use It for Life*. Simon & Schuster, reprint edition, January 2006.

[214] Peter D. Grünwald. *The Minimum Description Length Principle (Adaptive Computation and Machine Learning)*. The MIT Press, 2007.

[215] Yuan Li, Jessica Lin, and Tim Oates. Visualizing variable-length time series motifs. In *SDM*, pages 895–906. SIAM, 2012.

[216] Pavel Senin. Grammar-based time series classification with SAX-VSM. *University Of Hawaii, ICS CSDL Technical report*, 2014.

[217] P. Ordonez, T. Armstrong, T. Oates, and J. Fackler. Using modified multivariate Bag-of-Words models to classify physiological data. In *Data Mining Workshops (ICDMW), 2011 IEEE 11th International Conference on*, pages 534–539. IEEE, December 2011.

[218] P. Ordonez, T. Armstrong, T. Oates, and J. Fackler. Classification of patients using novel multivariate time series representations of physiological data. In *Machine Learning and*

Applications and Workshops (ICMLA), 2011 10th International Conference on, volume 2, pages 172–179. IEEE, December 2011.