Measurement and Analysis of Runtime-Metrics in a Continuous Integration Environment

Bachelorarbeit im Fach Informatik

vorgelegt von

Victor Simon

angefertigt am

Department Informatik
Lehrstuhl für Informatik 2
Programmiersysteme
Friedrich-Alexander-Universität Erlangen–Nürnberg
(Prof. Dr. M. Philippsen)

Betreuer: Andreas Kumlehn, M.Sc.; Prof. Dr. Michael Philippsen

Beginn der Arbeit: 30. Mai 2014
Abgabe der Arbeit: 27. November 2014
Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Der Universität Erlangen-Nürnberg, vertreten durch die Informatik 2 (Programmiersysteme), wird für Zwecke der Forschung und Lehre ein einfaches, kostenloses, zeitlich und örtlich unbeschränktes Nutzungsrecht an den Arbeitsergebnissen der Bachelorarbeit einschließlich etwaiger Schutzrechte und Urheberrechte eingeräumt.

Erlangen, den 27. November 2014

Victor Simon
Bachelorarbeit

**Thema:** Measurement and Analysis of Runtime-Metrics in a Continuous Integration Environment

**Hintergrund:** In a Continuous Integration environment, a software system under development is built from scratch, tested, and deployed after each change to achieve immediate feedback loops for software developers. Usually, the Continuous Integration server executes the unit tests and performs static source code analysis to detect possible errors in time. Besides avoiding such problems, the importance of a software system’s performance becomes increasingly important. The response time of a certain business case in a web application can, for example, change over the course of the development. Stress testing tools, e.g. Gatling, measure these response times by executing a reproducible load test that simulates a specified number of end-users following a protocol of interaction steps. Nowadays, the measured response times are typically neither recorded nor visualized during a build of the software system. The goal of this thesis is the implementation of a toolchain to automatically measure, visualize, and analyze the runtime metrics of a software system in a Continuous Integration environment. The automatic analysis of the measured time series uses Machine Learning approaches, e.g. Dynamic Time Warping, clustering algorithms, etc. to detect possible anomalies hidden in the software, e.g. memory leaks by examining the system’s memory usage. The results can help to draw the attention of the developers to possible problems and negative trends early during the development phase of a software system.

**Aufgabenstellung:**

- Getting familiar with the required tools: Continuous Integration server (e.g. Jenkins), stress testing tools (e.g. Gatling), virtualization software, e.g. Vagrant and Docker.

- Design and implementation of an automatic stress test toolchain in a Continuous Integration environment.

- Transfer of the measured results to a database, e.g. based on Solr, for further processing in tools like FindPerformanceBugs library and/or the Software-EKG.

- Reading about the relevant Machine Learning methods for the analysis of time series, e.g. Dynamic Time Warping, and selection of promising methods.

- Implementation of the previously selected methods in the FindPerformanceBugs library, based on the languages Java, R, Python, and/or Scala.
- Differentiation of the applied analysis methods as well as a documentation of their strengths and weaknesses.

- Processing and visualization of the results of the time series analysis in the Continuous Integration environment, e.g. via plugins.

- Documenting the results of the previously mentioned tasks in a scientific thesis.

**Meilensteine:**

- **Scientific:**
  - Concept and validation of the selected time series analysis methods

- **Engineering:**
  - Toolchain for measuring runtime metrics of software systems in a Continuous Integration environment
  - Implementation of the selected methods for the automatic analysis of the measured time series

**Literatur:**


- Amr: Survey on Time-Series Data Classification (http://goo.gl/L8Sfa3)

**Betreuung:**  Andreas Kumlehn, M.Sc., Prof. Dr. Michael Philippsen

**Bearbeiter:**  Victor Simon
Abstract

Nowadays, Continuous Integration frequently integrates and verifies software under development, in order to provide immediate feedback to the developers. This is accomplished by an automatic process that runs tests and measures static metrics, e.g. number of failed tests. However, to assess the dynamic behavior of software, it is necessary to measure and analyze dynamic run-time metrics, e.g. response time or memory usage. Although tools exist that are able to measure these run-time metrics, they are not easily automated and therefore do not integrate well in a Continuous Integration environment. Moreover, analyzing time series of run-time metrics with machine learning methods remains unattempted.

This thesis attempts to improve software verification by integrating stress tests and a dynamic machine learning analysis in a Continuous Integration environment. In order to achieve this, a toolchain is constructed that is able to automatically run stress tests and machine learning analyzes, store the results, and provide visual feedback inside a virtualized Continuous Integration environment. Furthermore, the new algorithm ATC is presented and evaluated, it combines the concept of time series motifs with k-means clustering and dynamic time warping to detect memory leaks from run-time metrics.

The evaluation results suggested a modified version that uses k-medoids clustering and a sliding window approach as more successful. Although ATC could not sufficiently detect memory leaks, the modified version provided more promising results. In conclusion, ATC needs to be evaluated further in regards to other applications and with more diverse example time series.
Abstract


## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Motivation</td>
<td>1</td>
</tr>
<tr>
<td>1.2 Overview</td>
<td>2</td>
</tr>
<tr>
<td>2 Foundations of Continuous Integration</td>
<td>5</td>
</tr>
<tr>
<td>2.1 Continuous Integration</td>
<td>5</td>
</tr>
<tr>
<td>2.1.1 Requirements of CI</td>
<td>6</td>
</tr>
<tr>
<td>2.1.2 Build</td>
<td>6</td>
</tr>
<tr>
<td>2.1.3 Tests and Inspections</td>
<td>6</td>
</tr>
<tr>
<td>2.1.4 CI Server</td>
<td>8</td>
</tr>
<tr>
<td>2.1.5 Continuous Delivery</td>
<td>8</td>
</tr>
<tr>
<td>2.1.6 Continuous Deployment</td>
<td>9</td>
</tr>
<tr>
<td>2.1.7 Course of action</td>
<td>11</td>
</tr>
<tr>
<td>2.2 Introduction to utilized software</td>
<td>12</td>
</tr>
<tr>
<td>2.2.1 Jenkins</td>
<td>12</td>
</tr>
<tr>
<td>2.2.2 Vagrant</td>
<td>13</td>
</tr>
<tr>
<td>2.2.3 Docker</td>
<td>14</td>
</tr>
<tr>
<td>2.2.4 Gatling</td>
<td>17</td>
</tr>
<tr>
<td>2.2.5 Solr</td>
<td>18</td>
</tr>
<tr>
<td>3 Implementation of a Continuous Integration toolchain</td>
<td>21</td>
</tr>
<tr>
<td>3.1 Construction of the toolchain</td>
<td>21</td>
</tr>
<tr>
<td>3.1.1 Environment</td>
<td>21</td>
</tr>
<tr>
<td>3.1.2 Build</td>
<td>23</td>
</tr>
<tr>
<td>3.2 Automating virtualization</td>
<td>24</td>
</tr>
<tr>
<td>3.3 Organizing the CI environment</td>
<td>25</td>
</tr>
<tr>
<td>3.3.1 Backup</td>
<td>25</td>
</tr>
<tr>
<td>3.3.2 Restore</td>
<td>26</td>
</tr>
<tr>
<td>3.4 Stress testing</td>
<td>27</td>
</tr>
<tr>
<td>3.5 Storing the data</td>
<td>29</td>
</tr>
<tr>
<td>3.5.1 Integration in Solr data schema</td>
<td>30</td>
</tr>
<tr>
<td>Section</td>
<td>Title</td>
</tr>
<tr>
<td>---------</td>
<td>-------</td>
</tr>
<tr>
<td>3.5.2</td>
<td>Processing stat files</td>
</tr>
<tr>
<td>3.5.3</td>
<td>Processing log files</td>
</tr>
<tr>
<td>3.6</td>
<td>Visualizing the results</td>
</tr>
<tr>
<td>4</td>
<td>Machine Learning</td>
</tr>
<tr>
<td>4.1</td>
<td>Machine Learning</td>
</tr>
<tr>
<td>4.1.1</td>
<td>Classification</td>
</tr>
<tr>
<td>4.1.2</td>
<td>Clustering</td>
</tr>
<tr>
<td>4.2</td>
<td>Time series analysis</td>
</tr>
<tr>
<td>4.2.1</td>
<td>Dynamic Time Warping</td>
</tr>
<tr>
<td>4.2.2</td>
<td>Time series motifs</td>
</tr>
<tr>
<td>4.2.3</td>
<td>Averaging of time series</td>
</tr>
<tr>
<td>4.3</td>
<td>ATC</td>
</tr>
<tr>
<td>4.3.1</td>
<td>Idea</td>
</tr>
<tr>
<td>4.3.2</td>
<td>Algorithm</td>
</tr>
<tr>
<td>4.3.3</td>
<td>Implementation</td>
</tr>
<tr>
<td>5</td>
<td>Evaluation</td>
</tr>
<tr>
<td>5.1</td>
<td>Evaluation strategy</td>
</tr>
<tr>
<td>5.2</td>
<td>Results</td>
</tr>
<tr>
<td>5.3</td>
<td>Alternative approaches</td>
</tr>
<tr>
<td>5.3.1</td>
<td>Cluster medoids instead of centers</td>
</tr>
<tr>
<td>5.3.2</td>
<td>Sliding window instead of motifs</td>
</tr>
<tr>
<td>5.3.3</td>
<td>Combining the approaches</td>
</tr>
<tr>
<td>5.4</td>
<td>Overall results</td>
</tr>
<tr>
<td>6</td>
<td>Conclusion</td>
</tr>
<tr>
<td>6.1</td>
<td>Future work</td>
</tr>
<tr>
<td>6.1.1</td>
<td>Improvements for the CI toolchain</td>
</tr>
<tr>
<td>6.1.2</td>
<td>Future work for ATC</td>
</tr>
<tr>
<td>6.2</td>
<td>Related work</td>
</tr>
<tr>
<td>A</td>
<td>Appendix</td>
</tr>
<tr>
<td>A.1</td>
<td>Specifications of the evaluation system</td>
</tr>
</tbody>
</table>
## List of Figures

2.1 A Java method that does not check for a *null* argument ........................................ 7
2.2 The deployment pipeline in Continuous Delivery ...................................................... 8
2.3 Course of action in Continuous Integration .............................................................. 10
2.4 A configuration element in the Jenkins web UI .......................................................... 12
2.5 Example for a Vagrantfile ............................................................................................ 14
2.6 Example for a script, written in the Gatling DSL .......................................................... 17
3.1 How the toolchain integrates in the regular CI process .................................................. 22
3.2 How the toolchain extends the regular CI build process (seen in the first rounded rectangle) ............................................................................................................. 23
3.3 Building the commands to reinstall Jenkins plugins via CLI ....................................... 26
3.4 Dockerfiles of the Main and Collector Images ............................................................... 28
3.5 Integration of Gatling results in the Solr schema ......................................................... 30
3.6 Example for a Gatling log file ....................................................................................... 32
3.7 Algorithm to calculate the user count as a time series ............................................... 33
3.8 Screenshot showing the different visualization elements ............................................ 34
3.9 Visualization of an FPB analysis .................................................................................. 35
4.1 Pseudo code for KNN ................................................................................................. 39
4.2 Pseudo code for k-means ............................................................................................. 41
4.3 Pseudo code for DTW ............................................................................................... 43
4.4 How FastDTW evaluates the warp path on different resolutions, with \( \text{radius} = 1 \). Taken from [73] .................................................................................. 45
4.5 Pseudo code for FastDTW, based on [73] .................................................................... 46
4.6 Shows the PAA (a) and the symbolic representation (b) (both taken from [51]) of a time series, and the lookup table (c) (taken from [45]) .................................................. 48
4.7 Pseudo code for DBA ................................................................................................. 50
4.8 Main algorithm of ATCTrain ....................................................................................... 53
4.9 Clustering algorithm of ATCTrain ............................................................................... 54
4.10 Algorithm to update the center of a cluster ............................................................... 55
4.11 ATCQuery .................................................................................................................. 56
5.1 \( \Delta r \) for ATC with different parameters ................................................................... 61
5.2 Result of ATC on two examples ................................................................................. 62
5.3 \( \Delta r \) for the *medoids* approach with different parameters ......................................... 63
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.4</td>
<td>∆r for the sliding window approach with different parameters</td>
<td>64</td>
</tr>
<tr>
<td>5.5</td>
<td>∆r for the combined approach with different parameters</td>
<td>65</td>
</tr>
<tr>
<td>5.6</td>
<td>Result of the combined approach on the examples from Figure 5.2</td>
<td>66</td>
</tr>
<tr>
<td>5.7</td>
<td>Colored examples for the result presented in Figure 5.6</td>
<td>67</td>
</tr>
</tbody>
</table>
1. Introduction

This chapter states why dynamic metrics are important and how they should be utilized in today’s software development. Furthermore, it provides a short overview over this thesis.

1.1. Motivation

Programming is difficult. With increasing complexity of modern software systems, programming errors become inevitable. Even worse, as the software “grows” around them, and the developer’s memory of the written code grows hazy, errors become harder to locate. As a result, errors are continuously more expensive to fix, the longer they stay undetected. Therefore, frequent verification is of tremendous importance in software development.

Nowadays, software under development is even built and verified after every change to the source code. This practice is called Continuous Integration (CI). After a developer commits his changes to the code repository, a CI server downloads the source code, compiles the software, and executes tests automatically. As a result, the CI server provides various static metrics, e.g. number of failed tests, or lines of code. Developers and managers then use these metrics to assess progress of the development process.

Although static metrics are useful, and give a lot of insight, they cannot represent dynamic behavior, e.g. how efficiently the software utilizes its resources. In order to get a more comprehensive view on the progress of development, dynamic run-time metrics, like memory, or CPU usage, are necessary. Tools exist that can measure run-time metrics of running software, e.g. JMX [37] for Java applications. When these metrics are measured over time, the values form a time series. However, time series can no longer be assessed as easily as static metrics. A human expert, usually a developer, needs to evaluate them, which takes a long time. Especially in more complex software that generates a high volume of measurements, this approach is impractical. In order to assess dynamic behavior, the human expert therefore needs assistance by the machine. Machine Learning is concerned with algorithms that are able to learn from experience to get gradually better at fulfilling a task. These algorithms can analyze previous data, e.g. time series, in order to predict, or classify future data. For instance, a machine learning algorithm could detect whether a software system allocates memory inefficient,
1. Introduction

based on previous memory usage of other systems. However, to accomplish this such an algorithm needs examples of inefficient and efficient memory usage, which would, again, require a human expert to provide them. On this account there is a need for another algorithm that can choose these examples by itself. As part of this thesis such an algorithm for automatic classification of time series is introduced.

In addition to the discussed metrics, stress tests simulate a load situation, and collect further run-time metrics, e.g. response time of a web application. While tools, like JMX, measure internal metrics of a system, stress tests provide external measurements. Together, they can supply developers with a pervasive view on the condition of a software project. Unfortunately, stress tests do not integrate well with CI. They often require manual activation, or modification. Furthermore, most stress test tools are designed to present their results to a human, hence generally provide them not in a machine-friendly manner. Because of these issues, executing stress tests is currently an overhead to the automatic CI process. This thesis aims to rectify this overhead, by integrating stress tests in a CI environment, which includes automatically running them, and storing the results, as well as visualizing them.

Nevertheless, reproducibility of the discussed metrics is crucial for their significance, meaning measurements of the same version of the software should lead to the same, or highly similar results. “Works on my machine” is a common problem in CI. Even though a developer generally builds the software, and runs tests locally before committing, the result on the CI server may still differ. That is because even small differences in software, or hardware configuration sometimes lead to largely different behavior, e.g. failing tests, or failing compilation. How to prevent this? A possible answer is Virtualization, where a resource, e.g. the CI server, is embedded in a virtual environment, e.g. a Virtual Machine that is portable, and behaves reproducible.

For that reason, this thesis attempts to combine the aforementioned methods to measure, and analyze run-time metrics in a virtualized CI environment that can be shared across developers.

1.2. Overview

First, Chapter 2 introduces CI and the most important tools used in this thesis. Section 2.1 presents the work flow of CI, commonly used terms, and improvements to the practice. Afterwards, Section 2.2 gives an overview over the most important features and advantages of tools that are utilized in the next chapter. Next, Chapter 3 describes the implementation of a toolchain that integrates the measurement and analysis of run-time metrics in a CI environment. While Section 3.1 discusses the thought process behind and the structure of this toolchain, the subsequent sections explain the particular step more thoroughly. The sections 3.2 and 3.3 describe how a
virtualized CI environment is set up and organized automatically, before sections 3.4 and 3.5 develop plugins that execute stress tests and store the results, respectively. Finally, Section 3.6 provides a method to execute machine learning analyzes and visualize the results, within the toolchain.

Consequently, this thesis covers machine learning in Chapter 4. In the first place, terms and popular approaches are described in Section 4.1. Moreover, Section 4.2 introduces selected algorithms for time series analysis. Afterwards, these algorithms are combined in Section 4.3 to construct a new algorithm for time series analysis.

Furthermore, the developed algorithm is evaluated in regards to its ability to detect memory leaks in time series in Chapter 5.

Finally, Chapter 6 provides a conclusion of the results and points out possible improvements for future work. Last, it presents a short overview over related work.
2. Foundations of Continuous Integration

In this chapter, the necessary foundations, regarding Continuous Integration, are explained, before they are utilized in the following chapters. Section 2.1 introduces the term Continuous Integration and discusses common processes, as well as related practices. Moreover, Section 2.2 presents the most important software, used in this thesis.

2.1. Continuous Integration

The term Continuous Integration (CI) was first introduced by Grady Booch \[9\] in 1991. Kent Beck \[7\] adopted it as part of his agile software development methodology Extreme Programming (XP). Martin Fowler summarized it, in an article \[20\] on his website, as follows:

CI is a software development practice where members of a team integrate their work frequently, usually each person integrates at least daily - leading to multiple integrations per day. Each integration is verified by an automated build (including test) to detect integration errors as quickly as possible. Many teams find that this approach leads to significantly reduced integration problems and allows a team to develop cohesive software more rapidly.

Until XP, the waterfall model was the state of the art in software development. In this model, integration of the components took place only at the very end of production leading to an “integration hell”, as referred to by Kent Beck \[7\]. The integration process uncovered errors not detected earlier, which had to be fixed under considerable time pressure to meet deadlines. Therefore techniques like Extreme Programming \[7\] and Continuous Integration attracted considerable attention, because of their radical difference to the established practice. Nowadays CI is still a popular discussed topic. As can be seen using the example of Continuous Delivery and Continuous Deployment, two practices still frequently written about \[33\] \[75\]. Section 2.1.5 and 2.1.6 explain these subjects.

Paul M. Duvall et al. wrote a book about CI and how to utilize it best \[17\]. They pursued the idea further, suggesting CI should not only include frequent integration,
2. Foundations of Continuous Integration

but also automation of the process. Meaning the integration happens as a result of the changes made to the software, instead of triggering it manually. This supports Fowler’s idea of making the integration process a “nonevent” that does not require any special effort.

The following explains CI more detailed, based on the aforementioned book [17].

2.1.1. Requirements of CI

According to Paul M. Duvall et al. [17], CI has four basic preconditions:

• A version control repository
• An automated build script
• A feedback mechanism, e.g. e-mail
• A process for integrating the source code changes, i.e. manual or with a CI server

(Manual integration processes are not covered in this thesis, see [17] for details.)

The integration process polls the version control repository continuously for changes, or gets notified. As soon as that occurs, it checks out the repository and starts to build the software. Afterwards one, or multiple developers receive immediate feedback about the result of the build. In case of a failed build, the system should report errors too. Generally feedback is the most important requirement, as well as value of CI. Without it, CI is pointless, as feedback allows the team to react quickly to complications.

2.1.2. Build

A build is a sequence of steps to transform source code into software, which works as a cohesive unit. The build consists of generating, testing and deploying the software. As a result a simple compile is not enough to classify as a build. The system needs to be tested, to verify it works as intended. Additionally the build can contain automated inspections as an additional tool to detect errors early. Finally the software has to be deployed to verify its integrity. A private build, is run by the developer, before he commits his changes, to ensure his code is not breaking the software. Private builds are often designed tentative, in order to run faster, while still catching drastic errors.

2.1.3. Tests and Inspections

As emphasized before, testing is an important part of the build process in CI. Code inspections can complement tests in verifying the code. While tests are a dynamic method, code inspections are static. Together they bring problems to the developer’s attention, early.
2.1. Continuous Integration

The most common tests in CI are unit tests. They test the functionality of a single unit of the program, e.g. a single class, or an individual method. Because they cover a minimal scope, unit tests should always run after the unit was compiled. As a result the build fails fast, when the error depends only on a single unit. Today there are many frameworks to write unit tests, e.g. JUnit \[42\] for Java programs.

Component tests are the next larger category of tests. They test larger parts of a program. Consider an online shop, built as a three-tier architecture. Component tests could exist for the logic, the front end and the communication with the database.

Finally system tests verify the functionality of the deployed application. In the case of the online shop, these could assert that a user is able to login, choose a product and buy it. Another form of system tests are stress tests. Stress tests simulate high load, e.g. a certain number of customers, on the application. A defined constraint of load capacity must be handled by the application to pass the test. For example such a constraint could demand that the online shop is able to serve 300 simultaneous customers with a maximum response time of 300 ms for each. Stress tests provide another safeguard, for the developers, to identify problems early. Unfortunately they are not common in CI systems.

Inspections can consist of manual code reviews and/or static code analysis. A CI environment can only put static code analysis to use, as the process must be automatable. Static code analysis can help to find formally provable bugs. This works on a simple principle.

The analysis software is configured with a predefined set of rules. When the software is supplied with source code, it tries to match its rules to the code. If a rule matches on a part of the code, this part typically violates a certain constraint. The build can then fail after the analysis, or just suggest the issue to a developer.

Consider the Java method in Figure 2.1. The return-statement in line 2 is error-prone. If the argument bar is null, this method will throw a NullPointerException. This behavior is most likely not intended. Static code analysis could detect this issue and indicate it to a developer. Furthermore static code analysis can enforce code practices. For example, to honor the hashCode-equals-contract in Java, the static code analysis could implement the following rule: “If equals() is implemented, hashCode() has to be implemented too”. SonarQube \[76\] provides this kind of analysis for multiple programming languages.

```
public boolean foo(String bar) {
    return bar.equals("baz");
}
```

Figure 2.1.: A Java method that does not check for a null argument
2. Foundations of Continuous Integration

2.1.4. CI Server

Although CI does not require a CI server, the former eases the process of developing a project with CI. If used, the CI Server is the centerpiece of a CI system. After a change happens, the CI server runs the build and provides feedback afterwards. Generally, feedback is the most fundamental argument for using a CI server. The developers can easily review feedback, e.g. test results, or statistics in a web interface. Furthermore the server can notify them about finished builds, or problems occurring. Further on, time stamps and build logs help to draw an even clearer picture about what is happening during the build.

There are countless implementations of CI servers, with different kinds of additional features, e.g. distributed builds over the network. Jenkins \cite{jenkins}, Hudson \cite{hudson}, CruiseControl.rb \cite{cruisecontrol} and Bamboo \cite{bamboo} are just a few examples. Jenkins is covered more detailed in Section 2.2.1.

2.1.5. Continuous Delivery

This practice was first elaborated in its foundations by Jez Humble et al. \cite{humble} in 2006. Eventually Humble and Farley established the term Continuous Delivery (CD) in their same-titled book \cite{humble2004continuous}. CD can be seen as thinking ahead of CI. While CI allows to bring software into a
deployable state at any time, CD aspires to be always able to release software into a production-like environment. However this does not mean the software is actually released to customers every time changes are integrated, though just the possibility to do so can simplify the release process to a button press.

Humble introduced a deployment pipeline, to achieve this. The sequence diagram in Figure 2.2, taken from [34], represents this pipeline. The first 2 steps, “Version control” and “Build & unit tests”, are basically CI. Automated Acceptance Testing (AAT) differs from CI. Although CD uses still the same kinds of automatic tests, as were used in CI, more effort goes into the tests. In CI, a test was often written by a single developer. Here however, tests are cross-developed across groups of multiple persons, including the developers, testers and customers. This extensive testing is done to make sure the software is as error-free as possible, before it moves towards the last two manual pipeline steps.

In User acceptance Testing (UAT), the software is deployed to users, who wish to test it. This step has two distinct purposes. On the one hand, users validate the quality of AAT. This embodies another safeguard to make sure automated testing is as reliable, as possible. On the other hand, the users can focus on aspects of testing a machine performs poor at, because regression testing is covered by AAT.

In the first place, these aspects include exploratory testing, the users explore the software, try different options, etc. Furthermore users may test the usability of the software in real world scenarios. Finally the look and feel of the software is tested, a task that is nearly impossible for machines, and hard for the developers, because they know how the software works and therefore cannot have a fresh impression of it.

Lastly, if UAT is approved, the software moves to the final step and actually is released to all users.

Note how every step in the pipeline contains a feedback loop, back to the delivery team. This emphasizes how feedback is even more important in CD, than in CI. Since the focus is no longer just on source code and bugs, but also tests, the tests need to be improved constantly. This is best achieved with extensive feedback, which users can provide in much more detail than machines.

Additionally the team behind the process is no longer a development team, but a delivery team. A delivery team consists of more people, involving, among others, developers, testers, operation staff, and managers. Essentially everyone involved in the release process should receive and act on all feedback available.

2.1.6. Continuous Deployment

Continuous Deployment is a variation of CD, introduced by Timothy Fitz [19]. The difference is that every build that passes AAT, gets released in a real production environment. At first, this seems like a very risky approach, though the risk can be reduced
by the following means.

Generally test coverage with AAT becomes even more important. Furthermore canary releases can reduce the risk involved. With a canary release, a smaller group of users receives the new revision immediately, and the behavior of the software is monitored for a certain time. If no problems occurred, the software gets rolled out to all users. Finally, an automated rollback process is able to interfere fast, if a released revision causes problems.

This approach is not reasonable for all applications but few, e.g. cloud services. According to Jez Humble [33], Google, Amazon, and Facebook practice Continuous Delivery.

Figure 2.3.: Course of action in Continuous Integration
2.1. Course of action

Figure 2.3 outlines the general CI process, skimmed over in Section 2.1.1. What follows explains this process in more detail with the aid of an example.

The developers of the three-tiered online shop, introduced in Section 2.1.3, practice CI using a CI server. Alice, a member of the team, has worked on the code and wants to commit her changes. She first runs a private build on her local machine, to prevent committing broken code. After the private build is successful, Alice commits her changes to the Version Control System (VCS). Subsequently the VCS accepts the changes and notifies the CI server about them. Since no build is currently running, the CI server starts a new build immediately. In the first place, it executes the automatic build script. The build script generates an executable and runs unit tests. Additionally, it runs component tests for the logic, the UI, and the database communication. After the build script is finished, the CI server deploys the application in a servlet container. Furthermore, it performs a stress test on the deployed online shop. The stress test spawns 300 customers and lets them browse the online shop simultaneously. Since no customer had a response time of more than 300 ms at any time, the test succeeds and a report, containing detailed results, is generated. Since all tests passed, the build is successful. Finally the CI server sends all developers an e-mail stating the VCS revision, the build duration, and the result.

Meanwhile, Bob, another developer, made changes of his own and ran a successful private build. He committed his changes shortly after Alice. As before, the VCS notified the CI server about the new changes. However Alice’s build was in progress at that time, so the CI server put Bob’s build in a waiting queue. As the former build is finished, the CI server now executes Bob’s build. Bob’s build successfully passes through the same steps, as Alice’s, but for the stress test at the end. The build fails. Furthermore the CI server sends an e-mail to all developers, informing them that the current build failed and needs to be fixed. Additionally it states that Bob was the author of the latest changes. Bob takes on the matter and opens the UI of the CI server. He looks over the stress test report, as well as the results of the other tests. Bob recognizes a component test in the logic took much longer than normal. After looking at the changeset of his commit, he discovers a bug in his code. He fixes the bug, causing the undesired behavior, and commits his changes. Afterwards the CI server runs another build. A few minutes later Bob receives an e-mail that the build was successful. Bob can now resume to his normal work.

This example displayed a typical scenario in the development with CI. An easy to fix bug was identified and fixed immediately, when the memory of the involved developer
was still fresh. However, without CI, this bug could have gone undetected until the final production phase, probably taking much more time to rectify.

### 2.2. Introduction to utilized software

This section gives an introduction to the most important software tools that are used in Chapter 3.

#### 2.2.1. Jenkins

Jenkins is a highly extensible CI server that run in any servlet-container. It is open source software and written in Java. According to its authors, it is designed to monitor the execution of repeated jobs with focus on two aspects. On the one hand it builds as well as tests software projects continuously to integrate changes into the project and detect errors early. On the other hand, it monitors executions of externally-run jobs. These jobs can run locally, or on a remote machine.

Java servlets, or just servlets, provide dynamic, component-based web services. Components are provided as a web archive (WAR) file and placed in a specific location. Then, a servlet-container, e.g. Jetty, Apache Tomcat, etc., provides a web server and dynamically serves a component when it is requested by a client. Commonly this is accomplished with a REST API, where the path on the website corresponds to the location of the web archive.

Since Jenkins runs inside of such a servlet-container, it is especially easy to install. Another important feature is the easy configuration that can be done entirely in a web UI, instead of editing XML files directly. Furthermore Jenkins is able to link builds to changes in a VCS and display them in the web UI. Jenkins also supports file fingerprinting that keeps track of the produced jars and dependencies of a build and is able to identify them. Distribution of builds or tests across multiple machines is possible as well.

Nevertheless, the extensibility via plugins is probably the most important feature.
This, in combination with its large and active community, provides Jenkins with enormous flexibility, as it can support nearly every tool that is used in CI. Although plugins can be developed in a multitude of different languages, e.g. Java, Python, Ruby, etc., this section only describes the plugin development in Java. Jenkins provides an API that defines extension points. An extension point is a Java class from which a plugin class can inherit, in order to include itself in the web UI.

For instance, the class Builder lets a plugin establish itself as a build step that can be added to a build via the web UI. When this build step is next in line, Jenkins calls the perform method in the builder class of the plugin and provides it with objects that can be used to access its API. The plugin can then fulfill its task and may even fail the build if problems occur.

In “.jelly” files, the plugin can then define various elements, e.g. checkboxes, tables, text fields, etc., which are configurable in the web UI. For example, the plugins that were developed as part of this thesis add configuration elements to the build configuration and the global settings as Figure shows for the gatling-docker plugin (will be described in Section 3.4).

Jenkins high market share of 70% across Java projects [39] puts it at an advantage against alternatives like Bamboo [6], Hudson [32], etc. This high usage, additional to its extensibility, makes Jenkins a prime candidate for a CI server to represent current usage of CI.

2.2.2. Vagrant

Vagrant [80] provides lightweight, portable virtual environments automatically. The project was founded in 2010 as a side project, before it formed into an open source project with commercial support in 2013 [80].

The virtual environments are organized in boxes that are easy to configure, shareable and disposable. Most notably, boxes behave consistently in every environment, which is especially useful to isolate configuration and dependencies from the actual software in development. Boxes are shareable through the service Vagrant Cloud [81] and can be started by a single command: vagrant up.

However Vagrant does not provide a virtual environment by itself, it just manages underlying providers. Such providers can be classic headless Virtual Machines (VMs) as well as other virtualization software, e.g. Docker (further details about Docker in Section 2.2.3). As providers virtualbox, VMware, Hyper-V, and Docker are supported. Additionally, Vagrant is extensible with custom providers. Due to the fact that most providers are able to use headless VMs, meaning no graphics are involved, these are very performant and as such can easily run in multiple instances on the same machine.

All necessary configuration of a box is managed in the Vagrantfile. Therefore only
2. Foundations of Continuous Integration

```ruby
# Vagrantfile API/syntax version. Don’t touch unless you know what you’re doing!
VAGRANTFILE_API_VERSION = "2"

Vagrant.configure(VAGRANTFILE_API_VERSION) do |config|
  # Every Vagrant virtual environment requires a box to build off of.
  config.vm.box = "phusion/ubuntu-14.04-amd64"
  config.vm.provision :docker
  config.vm.provision :shell, path: "scripts/bootstrap.sh"
  config.vm.network :forwarded_port, host: 28080, guest: 8080

  # Set the timezone to the host timezone
  timezone = File.read("/etc/timezone")
  puts 'Using host timezone: ' + timezone
  config.vm.provision :shell, inline: "echo \"#{timezone}\" | sudo tee /etc/timezone && dpkg-reconfigure --frontend noninteractive tzdata"
end
```

Figure 2.5.: Example for a Vagrantfile

this file is needed to port a box to another machine. Figure 2.5 shows an example. Line 8 states on which box the virtual environment should be based on. These base boxes are hosted in the Vagrant Cloud and can be installed automatically. For instance, the box configured in Figure 2.5 uses “ubuntu-14.04-amd64” from the user “phusion” as a base box.

Furthermore Vagrant features provisioners that alter the configuration of a box, e.g. install dependencies, include files from the host machine, etc., during a provisioning phase. This phase generally takes place directly after the box is started for the first time, though provisioning is also available through a separate command. First of all, the shell provisioner (line 11) executes a shell script on the guest machine. The most popular use case for this provisioner is to install dependencies on the VM and configure certain programs. Additional provisioners to add a file into the VM’s filesystem and more custom provisioners are available. Those custom provisioners often only install their respective programs on the machine as seen in line 10 with the Docker provisioner.

The ability to easily manage virtual environments lets Vagrant fill the gap between full-featured VMs, like virtualbox, and low level virtualization tools, like Docker. This thesis, therefore uses Vagrant to virtualize a CI environment.

2.2.3. Docker

Docker [14] encapsulates applications in virtual, portable containers, which can run on any newer system using a Linux kernel.

Docker is an open source project written in go and had its initial release in March 2013. The first stable version was available in June 2014.
2.2. Introduction to utilized software

Functionality

Docker works with a run-time environment that consists of a daemon which is accessed through a Command Line Interface (CLI). Docker uses multiple features of the Linux kernel to run a process in an isolated system. Most notable is the control groups (cgroups) feature. It controls resource usage, e.g. CPU, Memory, I/O, etc., of process groups.

Furthermore Docker utilizes a layered file system in combination with Copy-On-Write (CoW). When multiple coroutines use the same data, CoW means that the data is shared among them as long as no coroutine modifies it. If a coroutine accesses the data writing, the affected part of the data is copied and the change is written to the copy without notice of the coroutines.

As the name suggests, the layered file system enables mounting multiple file systems, with different access rights, on top of each other. This allows to share parts of the isolated system among multiple processes, which is the general idea behind Docker.

Layers  Docker mounts the root file system as read-only, followed by an additional file system with read and write (r/w) rights. Each of those file systems is called layer. Multiple layers can be stacked on top of each other. If a process creates, removes, or modifies a file, another layer is added. Respectively, Docker creates the file, hides the file, or modifies a copy of the file in the new layer. If two layers contain the same file in different versions, each process can see only the version of the top most layer. This allows Docker to manage systems in a VCS-like way.

Images  An image is a read-only layer. As such, it never changes. Each image can have one or more images underneath it, the parent images. An image with no parent is called base image. Docker can tag images for better overview.

Containers  A container consists of an image with an additional r/w layer and metadata, e.g. unique identifier or network configuration. Containers start with a single process, though this process can start other processes. Containers can be started, stopped, or restarted.

Since a container is not immutable, it has a state. The state includes its r/w layer, whether the container is running or exited, the processes running inside (if running), and the exit value (if exited). Each container is labeled with a unique identifier, either generated or specified by the user. Additionally Docker can turn containers into images with docker commit.

Building images  The user can build a custom image by modifying containers and committing them to an image, though this manual process can be tedious. Dockerfiles automate this task. Each Dockerfile starts with a base image to build from, followed by an optional maintainer. Thereafter, different steps can be specified. The
most common include, run a command, add a file to the container, mount a volume, expose a port, etc. 

**docker build** processes each step of the Dockerfile and runs a container with the corresponding command. After the container exits successfully, Docker commits it to a temporary image. This image is then used to create a container for the next step and deleted afterwards. Docker stops the building process if the container fails.

**Running containers**  **docker run** creates containers and starts them, taking an image and a command list as arguments. An image may define an *entrypoint*, a program that is called when the image is run in a container. For such an image the command list is delegated, as arguments, to the entrypoint. Generally, Docker creates a container from the image and starts it. Nevertheless additional features are available. A few selected are listed below:

- Map ports from the host system to the container.
- Map directories from the host system, as volumes, into the container.
- Inherit volumes from another container, including modifications made by the other container.
- Link containers with each other in a parent-child-relationship, allowing the parent to communicate with the child over network.

**Sharing** Images and containers are fully serializable. Docker supports importing and exporting them via data streams. Additionally Docker offers Dockerhub, a cloud service, where users can share their Docker images.

**Benefits**

Docker comes with certain traits that are desirable in a CI environment. First of all, Docker provides an unprecedented scalability. While regular virtualization means need a significant overhead for each instance, Docker gets along with a minimal overhead and scales considerably well. Further on, Docker containers are reproducible, because they always start based on the same, immutable images. Furthermore Docker manages to isolate an application. Isolation applies to dependencies as well as the host system. Additionally, Docker is portable. Containers can run on any Linux system as well as any other system thanks to boot2docker, a lightweight VM specifically developed to run Docker. Finally, development teams can put their Dockerfiles under version control and track changes of their images.

In addition a remote API for Docker is available. It uses JSON to transfer data and HTTP for communication. There are integrations of this API for multiple programming
2.2. Introduction to utilized software

Figure 2.6.: Example for a script, written in the Gatling DSL

language, e.g. Python, Java, and Groovy.

In general, Docker is a young project with an astonishing history. After its first release, the project became a highly discussed topic, leading to a fast growing community. Even before its first stable release, Docker was used by significant companies like Ebay [18]. This thesis uses Docker for low-level virtualization, because it provides an efficient way to encapsulate software from dependencies and data.

2.2.4. Gatling

Gatling [26] is an easy to configure stress test tool that features its own Domain Specific Language (DSL). It is a rather young open source project that had its first stable release in 2013 and recently released version 2.0. Gatling is written entirely in Scala and licensed under the Apache License.

On the basis of the example script in Figure 2.6, the following explains the most important features of the Gatling DSL. The full version was used to stress test Intelligent...
2. Foundations of Continuous Integration

*Home*, a test application that organizes devices using a REST API. Each Gatling script begins with some necessary imports (lines 1-3). Furthermore, every Gatling script, is contained in a class that inherits from *Simulation* (line 5). This class is the stress test and is referred to as *simulation*. The lines 8-14 initialize the HTTP protocol and establish a base URL, i.e. a target for the simulation.

Next, a *scenario* is created that typically represents the behavior of a user or a user group, e.g. the scenario in line 16 registers a device. Scenarios consist of *records* and pauses. A record type has a *record type*, e.g. “getRoot” in line 17, that later occurs in a generated HTML report and usually executes a request, e.g. line 18 executes a get request on “172.17.42.1:8086/”. Additionally, records can perform checks on the response of the target and many other actions.

Line 21 shows the flexibility of the Gatling DSL. The script retrieved a value from the JVM in line 7 that is now used to construct an URL, which registers a device in the application. Afterwards, the script loads the recently added devices from the application and checks if its devices were actually registered (lines 23-25).

Finally, the simulation is configured with the *setUp* function that generates users, which execute scenarios, e.g. line 27 generates one user in one second.

This script was used to test the plugin that is developed in Section 3.4. The plugins starts Docker containers that run this script in order to register multiple devices with unique names.

Every Gatling simulation is a Scala program and therefore easily modifiable in hindsight. Gatling can generate multiple users and assign them with certain behavior via scenarios. Additionally, it can feed data to the users in multiple ways, e.g. CSVs or generators. Furthermore, Gatling also features a GUI tool to record simulations.

In the back end users are managed by *AKKA* [1] and *Netty* [63], two frameworks for building asynchronous distributed applications within the JVM. As a result computing resources are handled more flexibly than with a single-thread-per-user approach.

This thesis utilizes Gatling as a stress test tool over alternatives like *HP Load Runner* [30] or *JMeter* [2] since it is free and provides an easy to use DSL.

### 2.2.5. Solr

*Apache Solr* [4] is a fast, document oriented enterprise search engine platform running inside a servlet container. It is an open source software, written in Java and developed by Apache.

**NoSQL**  *NoSQL* is an approach to store data in other ways than with the traditional relational model. It is often interpreted as *Not Only SQL*, because NoSQL does not necessarily omit *SQL*.

Simplified, a NoSQL database can be seen as having only one table that contains all the
columns. This provides better horizontal scalability across large datasets as well as an easier to understand design.

**Apache Lucene**  
*Apache Lucene* is a text-based search engine library developed by Apache. It allows storing data, called *indexing*, and searching it again. Lucene organizes data in so called *documents*, sets of key-value-pairs. A *schema* states of which keys, the *fields*, a document consists of.

Solr is built on top of Lucene, providing a front end for the library and extending it by its own features. Along performance optimizations, Solr features *faceted search* and *html administration*. On the one hand faceted search enable to filter and organize data using any combination of fields in the schema. The html administration, on the other hand, provides easy access to Solr. It shows information about resource usage and provides a web based search. Additionally, it features a REST-like API to search and index documents that can transfer data in various formats, e.g. XML, JSON, CSV, etc.

In the first place, the NoSQL approach makes Solr a performant and scalable solution to store and retrieve large chunks of data. Furthermore, it provides a flexible way to view data with faceted search. Additional flexibility is offered when deploying Solr. Since it runs in a servlet container, the user can choose which server he wants to use with it. Finally, the HTML administration makes Solr easily, as well as automatically, accessible.

According to *DB-Engines* [13], Solr is the most popular enterprise search engine. In addition, faceted search and the NoSQL approach make it well suited for time series data. Faceted search allows to store the actual data in a single field and information about the data in the other fields. Since the documents are in a single table, Solr only needs to filter the information fields and deliver the data field. As a result, access times are very fast.

Since an environment that is able to store and retrieve time series using a Solr instance was already available, this thesis utilizes this environment and Solr.
3. Implementation of a Continuous Integration toolchain

This chapter describes a toolchain that integrates Gatling stress test and machine learning analyzes in CI. Section 3.1 constructs the toolchain and explains the responsibility of each link. Then, Section 3.2 explains how a CI environment can be set up with Vagrant, before Section 3.3 describes how the configuration of this environment can be saved persistently and restored.

Moreover, the following sections cover the build process in the environment. Section 3.4 develops a plugin that uses Docker and Gatling to run stress tests. In Section 3.5, another plugin is developed that is able to collect the results of the previous plugin and store them in a Solr instance. Finally, Section 3.6 executes machine learning analyzes and shows how the results of the stress tests, as well as these of analyzes, are visualized inside the CI environment.

3.1. Construction of the toolchain

A large part of this work relies on building a Continuous Integration toolchain to integrate stress tests into a CI environment. This Section constructs such a toolchain in two steps. First a virtualized CI environment, including a CI server, is set up automatically. Afterwards, a build order is established. The build order includes the execution of stress tests, storing of the results, as well as analysis of externally collected run-time metrics. Additionally it visualizes the results of the stress tests, as well as the analysis, in order to provide immediate visual feedback after the build.

3.1.1. Environment

Figure 3.1 shows how the CI environment integrates into the general CI process, shown in Figure 2.3. Revisiting the aims set in Section 1.1, it has to fulfill two requirements. On the one hand it has to be virtualized, to ensure portability of the test environment and guarantee reproducibility. On the other hand it needs to be set up automatically. Automation prevents the error-proneness that comes with executing multiple manual steps. As a result every developer of the team should be able to set up the environment and execute a private build, identical to one that the central CI server would have executed.
Virtualize the CI environment

Vagrant is used to set up a VM, containing the entire Continuous Integration environment. As discussed in Section 2.2.2 all necessary configuration is set up in a Vagrantfile. This is done for portability reasons, as the Vagrantfile can be checked into a VCS. As part of the configuration, all dependencies needed, including the CI server Jenkins, introduced in Section 2.2.1 are installed. Furthermore Vagrant forwards a port to the host machine, to enable communication with Jenkins from outside the VM.

Automate set up

Although the VM is set up automatically in the former step, the CI environment is not. Since Jenkins is freshly installed on every new instance of the VM, configuration does not persist. To achieve this, a further step is necessary. Jenkins’ data is saved separately and put under version control like the Vagrantfile. Every time a developer creates a new VM, the configuration is loaded after the boot process.
3.1. Construction of the toolchain

Figure 3.2.: How the toolchain extends the regular CI build process (seen in the first rounded rectangle)

3.1.2. Build

The major part, the toolchain, is integrated in the build process. Figure 3.2 illustrates how the toolchain extends the general CI build, seen in Figure 2.3. In the course of this section, the shown workflow will be developed, using the tools introduced in Chapter 2. Section 1.1 formulated the following tasks for the build:

- **Run stress tests:** By integrating the stress tests in the build process, regular performance testing is enforced. So malicious changes, affecting performance, are detected immediately.

- **Store the measured data in a database:** Storing the measurements consistently, makes them easily accessible for consecutive steps.

- **Analyze run-time metrics:** As the stress test provides an outside view to the system, run-time metrics provide the same from the inside. This helps to draw a clearer picture of the software’s performance.

- **Visualize the results:** Visualizing the measurements as well as the analyzes, supplies the developer with an immediate feedback loop, to assess the impact of changes made.

**Run stress tests**

First of all Gatling, described in Section 2.2.4, is combined with Docker, described in Section 2.2.3, in order to automatically run stress tests. This is accomplished by developing a Jenkins plugin. The plugin distributes a Gatling simulation across one
or multiple Docker containers. First, it creates containers to execute a simulation and create log files. Afterwards, it uses another Docker container to collect the results and generate an HTML report.

**Store the measured data**

After the log files and the reports are available, the data has to be stored in Solr, described in Section 2.2.5. Solr files the run-time metrics, used in a later step as well. To store the measured data in Solr, a second Jenkins plugin is developed. The plugin selects, and parses the relevant files. Furthermore it arranges the data into the Solr schema, since the schema is already established to organize other run-time metrics. To fit into the schema, the plugin may need to be select, transform, or summarize the data.

**Analyze run-time metrics**

*Find Performance Bugs* (FPB) [72] is a software analysis tool for time series. It retrieves time series from Solr, and analyzes them. The results are provided in the form of a CSV file. FPB will be extended with a new analysis, as part of this thesis, and therefore revisited later, in Section 4.3.

**Visualize the results**

As the last step, the results of the build are visualized to provide immediate feedback. The results consist of the report generated by Gatling, as well as the CSV file generated by FPB. A *Gatling plugin* [25] for Jenkins exists that links Gatling reports on the project page. In addition, a Python script plots the CSV file and generates an HTML document. Afterwards, the HTML Publisher plugin displays the generated document.

### 3.2. Automating virtualization

As described in Section 3.1, an automated virtualized CI environment provides reproducibility and portability. All necessary data is stored in the *vagrant* folder. This is a special folder, which is available from inside the VM under `/vagrant`. It is put under version control, therefore the machine is easily portable.

Configuration of the VM happens in two steps. On the one hand, the Vagrantfile contains configuration that is required to happen there, or simply easier to do there. The shell provisioner, on the other hand, handles the rest of the configuration.

The Vagrantfile, used here, was already shown in Figure 2.5. As a base box a special Ubuntu image, provided by *Phusion* [68], is used. Phusion have adjusted the base Ubuntu image for Vagrant to work better with Docker. Along these adjustments, they allocate more virtual CPUs by default, install a more recent Linux kernel, enable the
3.3. Organizing the CI environment

cgroups-feature, etc. The Docker provisioner, built into Vagrant, installs Docker. This is the easiest way to install Docker, as it is accomplished in a single line of code. Therefore, this is not done with the shell provisioner. Afterwards, Vagrant calls the shell provisioner, described later. To enable access to the web interface of Jenkins from the host port 8080, on the VM, is forwarded to port 28080, on the host. The Vagrantfile uses a higher port to avoid collisions with other applications. Additionally, port 8086 is used, to forward the port of a test application to the host. Finally, Vagrant injects the time zone from the host into the VM. This is done, since Vagrant uses a default UTC time for all VMs. As a result, Jenkins would show the wrong time without this step.

The shell provisioner executes a bootstrap shell script to handle the rest of the configuration. This script is executed, with root rights, on the freshly built VM. As the script should run automated, the Debian front-end is set to non-interactive. This prevents programs, like apt-get from displaying dialogues on the console. Since Jenkins requires Java, Java 8 is installed as the next step, before Jenkins itself is installed. The script additionally installs git\textsuperscript{[27]}, to provide a VCS for the environment and additional dependencies, e.g. Python libraries. Furthermore, it copies the Docker configuration file from the vagrant folder on the host, and adds the User jenkins to the Group docker, e.g. to start a servlet container as a target for stress tests. Finally, the script creates a folder, owned by the User jenkins, to buffer configuration backups in.

3.3. Organizing the CI environment

Jenkins’ configuration is retained outside of the VM. Jenkins stores all configuration files in the JENKINS_HOME Directory, defaulting to /var/lib/jenkins. A Python script allows the data to be backed up or restored, on demand, from the host machine. Making the backup/restore process on demand, as opposed to automated, gives the developer more fine-grained control over updates to the configuration. He can test a new configuration and save it, or simply restore the old configuration, without creating a new VM. As backup and restore are managed by a single script, this manual step creates acceptable overhead, for its benefits, with respect to the automation of the environment. The Python script setupjenkins takes a single argument, which can either be backup or restore. It opens an ssh connection to the VM and executes the corresponding script to its argument. These scripts are projected into the VM through the vagrant folder, as before.

3.3.1. Backup

For a naive backup, JENKINS_HOME could simply be mapped into the vagrant folder, in order to save the data. Though, while most configuration consists of cheap to store XML files, larger files are also stored there, e.g. plugins. Since plugins are easily obtainable
3. Implementation of a Continuous Integration toolchain

```python
# build plugin commands
pluginxml = jenkins_home + os.sep + 'installedPlugins.xml'
plugins = et.parse(pluginxml).getroot().find('plugins')
plugincmds = []
cmd = clicmd + ' install-plugin '
for plugin in list(plugins):
    pldata = [x.text for x in list(plugin)]
    if 'Hudson' in pldata[0]:
        continue
    if 'dfd-solr-jenkins' in pldata[0] or 'gatling-docker-plugin' in pldata[0]:
        plugin_url = thesis_plugins_url + pldata[0] + '.hpi'
    else:
        plugin_url = jenkins_plugins_url + '/'.join(pldata) + '/' + pldata[0] + '.hpi'
    plugincmds.append(cmd + plugin_url)
```

Figure 3.3.: Building the commands to reinstall Jenkins plugins via CLI

from the Jenkins repository, they should not be backed up. There are many Jenkins plugins, designed to back up configuration. For this purpose, ThinBackup [79] was chosen. ThinBackup is able to retrieve names and versions of all installed plugins from Jenkins and store them in an XML file. Furthermore, it can initiate a backup via Jenkins’ REST API.

A shell script triggers a backup, by downloading a specific URL. The backup is stored in the backup folder, allocated earlier. Additionally, the Jenkins CLI is downloaded to make the restoring process of the plugins easier. Finally, the data, including the CLI, is packed into a tar archive and saved in the vagrant folder.

3.3.2. Restore

To restore a backup, another Python script is used. First of all, it stops the Jenkins service to ensure a clean restoration. Then, it empties JENKINS_HOME. As a result, the backup can be safely unpacked and its contents copied. ThinBackup stores information about the installed plugins in the XML file installedplugins.xml. The script extracts this information and builds commands to install the plugins with the Jenkins CLI. An excerpt from the restore script, that accomplishes this, is shown in Figure 3.3. The whole script is omitted for brevity.

Line 3 parses the XML and stores the names and versions in a list. In lines 6-17, the script iterates over the plugins and builds commands for the Jenkins CLI. Plugins that are part of this thesis are not available from the official Jenkins repositories and therefore need to be filtered and retrieved separately, as shown in line 12f. After all commands are built, the script deletes the XML file, as it is no longer needed. Because the script is executed as root, the ownership of JENKINS_HOME, and its con-
3.4. Stress testing

The gatling-docker plugin uses docker-java \[15\] to communicate with the Docker daemon over TCP. Docker-java is a client implementation for the Docker remote API. In order to use the remote API, Docker needs to be configured to listen on a TCP port. To accomplish this, the Docker daemon is started with the -H option and a TCP socket address, e.g. 
\[-H tcp://127.0.0.1:2375\] to listen on port 2375.

The user has to manage this configuration by himself. In this thesis, this was accomplished in Section 3.2, by copying a Docker configuration file during the provisioning of the VM. Additionally the Docker daemon needs to use the same version of the API, as the client implementation. Though this is easily achieved, since both the daemon and docker-java are backwards compatible with lower versions.

The user configures the TCP address of the daemon, and the client version in the global Jenkins configuration as it was already shown in Section 2.2.1. In the build step, the user can set a simulation, and a report folder. Additionally, a scale factor controls over how many containers a single script is distributed. A checkbox controls, whether multiple simulations are executed in parallel, or sequentially. These features will later be elaborated further. Finally, another checkbox determines, if the output of the containers is printed directly to the Jenkins console. The last option is mainly useful for debugging purposes, as the output can get rather lengthy, and is saved with the reports via log files anyway.

Before every run, gatling-docker ensures his configuration is correct, by pinging the Docker daemon. If it cannot reach the daemon, it fails the build and notifies the user to check the configuration. Otherwise, it creates the report folder, if it does not exist already. This is done to prevent Docker from creating the folder later, making it owned by root and hard to remove. Next, the simulation folder is scanned for Scala files, which are assumed as Gatling simulations. Additionally, the user id of Jenkins is collected with the Linux program id, as it is needed later. Afterwards, the plugin pulls two necessary Docker images from the Docker registry. As a result, the first run of the plugin takes a long time compared to consecutive runs. Nevertheless, these images have largely identical layers which creates only negligible overhead to download the second image. Eventually, the plugin can run the detected simulations and map the generated reports
3. Implementation of a Continuous Integration toolchain

```bash
FROM ubuntu:14.04
ENV DEBIAN_FRONTEND noninteractive
RUN apt-get -qy update 
  && apt-get install -qy openjdk-7-jdk python curl unzip 
  && apt-get autoremove 
  && apt-get clean 
# get gatling
RUN curl http://repo1.maven.org/maven2/io/gatling/highcharts/gatling-charts-highcharts
   /2.0.1/gatling-charts-highcharts-2.0.1.bundle.zip -o /opt/gatling.zip 
  && unzip /opt/gatling.zip -d /opt/ 
  && rm /opt/gatling.zip 
  && mv /opt/gatling-charts-highcharts-* /opt/gatling 
VOLUME /gatling-simulations 
VOLUME /gatling-results 
VOLUME /gatling-report 
ADD transformsimulation.py /opt/gatling/bin
ENTRYPOINT ["python", 
             "/opt/gatling/bin
             
transformsimulation.py"]
ADD generatereports.py /opt/gatling/bin
ENTRYPOINT ["python", 
             "/opt/gatling/bin
             
generatereports.py"]
```

(a) Main Image

(b) Collector Image

Figure 3.4.: Dockerfiles of the Main and Collector Images

back to the host file system. After the simulations finish, it removes the used containers to save disk space. Since all files that Docker creates on the host system are owned by root, a throw away container changes the owner of the report folder to the user id obtained earlier. Finally, gatling-docker closes the connection to the Docker daemon.

As stated before, the two Docker images, and therefore the Dockerfiles are largely identical. Figure 3.4 shows these files. They are based on Ubuntu 14.04 and install Gatling, as well as necessary dependencies, e.g. Java, for it. Commands are concatenated to keep them in a single layer and save space in the image. Since Docker creates a new layer for every command, temporary data would otherwise be included in the Docker image. Additionally, the files establish three volumes for the simulation files, the results, and the report, respectively. Beginning at line 20 in Figure 3.4, the Dockerfiles start to differ. They both add a script, and establish it as an Entrypoint to fulfill different tasks. An Entrypoint enables the image to be run like an executable file. When the container is started, all arguments are delegated to the Entrypoint. This provides a certain comfort, and better code visibility, when working with Docker.

The Main image 3.4(a) is called with a simulation name and a container id. It looks for the corresponding Scala file in the simulation folder, and starts the simulation with
3.5. Storing the data

Gatling. At the same time, it injects a variable \texttt{container.id} via the \texttt{JAVA_OPTS} environment variable. This container id can be used in the simulations to recognize the container that executes the simulation. Gatling is started with the \texttt{no-reports} (-no) option, in order to generate log files only.

The \texttt{Collector} image \texttt{3.4(b)} can then use the log files of all containers to generate a report for the simulation. Since Gatling names all log files the same, they are appended with a number and copied to the report folder that is mapped to the host file system. Afterwards, Gatling is started with the \texttt{reports only} (-ro) option to generate a report. Both images exit with the same \texttt{exit status} as Gatling. The images described here are publicly available at the Docker registry, under \texttt{vsimon/gatling} and \texttt{vsimon/gatling-collector}, respectively.

The list of simulations is processed either sequentially or in parallel. This enables the plugin to simulate multiple situations at the same time, which is desirable to emulate a more realistic environment. Each Docker container is represented by a Java object of the class \texttt{DockerContainer}. An instance of \texttt{DockerContainer} can create, remove, start, or wait for, a container. In the first place, every simulation receives a data container, which is used to transport and encapsulate data between containers. The data container is based on the \texttt{busybox} image, a very tiny Linux image that contains only the most basic functionality. The plugin creates a number of Main containers, equal to the configured scaling factor, linking every container’s results folder into the data container. It starts the containers and waits until the simulation is complete. After that, the plugin starts a Collector container. On the one hand, the Collector is linked to the data container that contains the log files. On the other hand, the report folder on the host is mapped into the Collector as well. When the Collector is finished, the report is available at the host file system. This procedure is repeated for every simulation found. However all the files in the report folder belong to root, and need to be transferred to another user id. This is accomplished with the Linux program \texttt{chown}.

Additionally, the plugin saves the output of all containers as log files in the report folder. Finally, it removes all used containers, and prints a message stating the location of the log files.

3.5. Storing the data

After the stress tests are executed, the results are stored in Solr. Gatling generates HTML reports, as explained in Section \texttt{2.2.4}. The relevant information is stored in two ways. On the one hand, each report contains a Javascript file. This \texttt{stat file} contains global statistics about the simulation, e.g. mean response time. These statistics are also available for each \texttt{record type} separately.

On the other hand, Gatling generates log files during the stress test, which contain
3. Implementation of a Continuous Integration toolchain

Figure 3.5.: Integration of Gatling results in the Solr schema

raw data about the simulation. Here, each Docker container used by the *gatling-docker* plugin generates one log file. The data is available in the form of a pseudo TSV file. The first line contains general information on the simulation, e.g. name, starting time. Meanwhile, the following lines contain information on requests, users, etc. For instance, Figure 3.6 shows a log file.

The Solr instance already stores run-time metrics in the form of time series. Therefore, the available data needs to be selected and integrated in the established data schema, before it can be stored. For this purpose the *dfd-solr-jenkins* plugin was developed. An URL, for the Solr instance, can be configured, in Jenkins’ global settings. Moreover, in the build step, dfd-solr-jenkins expects a folder, which contains Gatling reports. Additionally, there is an option to set an alternative directory from which the Git revision is retrieved, which is needed later. By default, that is set to the *Workspace* of the build. The plugin expects that each report, is in its own folder and contains exactly a single stat file.

First, it scans the report folder, for stat files and processes the associated reports separately. Afterwards, it prints how many time series were successfully added to Solr. The plugin marks the build successful, unstable, or failed, depending if all, a portion, or no time series were added, respectively.

The following Sections explain, how the available data is integrated into the established schema, how stat files, and how log files are processed.

### 3.5.1. Integration in Solr data schema

As mentioned, the Solr instance, used for storing the Gatling results, already has an established data schema. How can the available data be selected to fit in this schema? Since the schema is used for storing time series, the data has to be transformed to represent a time series. Log files can easily be represented as a time series, as they contain
3.5. Storing the data

The plugin extracts 3 time series from a log file: the request time, the request status, and the count of active users per second. However, the stats file contains only single values for each report and record type. To represent this as a time series, it adds new values of a report and a time stamp of the current time to possibly existing values in Solr. With this approach, a time series is built over multiple reports.

Figure 3.5 shows the existing schema and how it is applied for this purpose. Series is the top most category. To distinguish Gatling time series from other series, this is simply a constant string.

However, storing the actual host, where the test was executed, in Host, would be of little benefit, since it does not affect the result. Other candidates were the Jenkins build number and the Git revision. As the build number does not necessarily distinguish between different versions of the application, the Git revision was chosen as most appropriate.

Process simply contains the name of the simulation.

For Group, the plugin distinguishes, based on the source of the information. While log files contain the scenario name, the stat file does not. The plugin therefore sets group to the scenario name, or “Global”, respectively. This has the benefit, that the results can be grouped, depending on their origin.

Measurement is organized in a similar manner. It contains the record type, or “Global Information”, which holds the concatenated data for all record types of the simulation. As this specific string is already used by Gatling in the stats file, it is maintained here, for simplicity.

Finally, in Metric, the plugin stores the name of the time series, e.g. requesttime, for the duration of each request, or meanResponseTimeOK, for the mean response time of all successful requests.

3.5.2. Processing stat files

The stat file is a Javascript file, which is used by the HTML report. Information is stored in a global variable stats. The plugin extracts the contents of this file with the Scripting-Engine Rhino [71].

First, it evaluates the file, before it extracts the relevant variable. Then, the stats variable is converted to a JSON string, for further processing with Gson [29]. The JSON object contains the fields stats and contents, on the highest level. Stats consists of another JSON object containing the Global Information. Meanwhile, in the lower levels, contents contains a Javascript object for each record type. This object itself, subsequently also holds a stat field. The plugin extracts all stats fields from the JSON and uses Gson to construct a POJO (Plain Old Java Object) from each.

As the next step, the plugin queries Solr for, potentially existing, time series of the current simulation. The result of the query is sorted into a two dimensional map, depending on the Measurement, as first, and the Metric, as second dimension. Further on, each
3. Implementation of a Continuous Integration toolchain

3.5.3. Processing log files

An example for a log file, as is discussed here, is shown in Figure 3.6. Log files are read using Super CSV \[78\]. The first line of each file contains the name of the simulation, an information not present in a stat file. Therefore it is important to process the log files first, before processing the corresponding stat file. The remaining lines of all log files of the report are sorted into two lists, depending on the Action column, the third column in Figure 3.6. Lines with the Action “USER” contain information about a simulated user, while those with the Action “REQUEST” contain information about a request. Other values for Action, e.g. “GROUP”, are ignored.

Each request line contains four time stamps: the first byte sent (\textit{FBsent}), the last byte sent (\textit{LBsent}), the first byte received (\textit{FBreceived}), and the last byte received (\textit{LBreceived}). Since the log files were created simultaneously, they need to be sorted, after the first byte sent, first. Storing all four time stamps would deem to extensive for a clear view on the information. As of this reason, the plugin considers \textit{LBreceived} – \textit{FBsent} as the request time, and \textit{FBsent} as the start of the request. Furthermore, a request line includes a status column, containing either “OK”, or “KO”. Since the resulting time series should contain real numbered values, “KO” and “OK”, are converted to, 0 and 1, respectively.

After the preprocessing is done, the lines are sorted into a two dimensional map, as already used for preprocessing a stat file in the previous section. Then, the plugin can create data points for a status and a request time series, using the start of each request as time stamp. Afterwards, the map is flattened to create a list of time series for Solr.

POJO tries to locate itself in this map. If it does not find the Measurement/Metric combination for an attribute, it creates a new time series entry for this attribute. Which attributes are transferred to Solr, is hard coded in a static String array. The POJO uses Java’s expression API to access the corresponding attributes. Finally, the map is flattened and the resulting time series are added to the Solr instance.
3.6. Visualizing the results

```python
def calculateUserCount(users):
    # sort after start date
    sort(users)
    start = users[0].start
    end = users[len(users) - 1].end
    # Array of size secondsBetween(start, end) with tuples (count, time stamp)
    userCount = array(secondsBetween(start, end))
    for user in users:
        offset = secondsBetween(start, user.start)
        duration = secondsBetween(u.start, u.end)
        for i in range(0, duration):
            userCount[offset + i] = (userCount[offset + i][0], start + offset + i)  # count
                                                # time in seconds
    return userCount
```

Figure 3.7.: Algorithm to calculate the user count as a time series

However, creating a time series of the user count at any given time is not as straightforward. The user count is not recorded in the log file, but only the starting time and the end time for each user’s time window. User lines contain two time stamps and a constant. The constant is either “START” or “END”. For start lines, the first time stamp states the starting time of that user, while the second time stamp is set to zero. For end lines, the first time stamp is unchanged, but the second time stamp marks the end of that user’s time window. Using these lines, the plugin creates an object for each user, holding its start and end time. These objects are sorted in lists, corresponding to their scenario, and in a list for all scenarios.

The algorithm to calculate the user count on a per second basis, is shown in Figure 3.7. It takes a previously constructed list as input and creates a list of data points for the time series. After the calculation, the time series is added to the other time series and stored into Solr.

3.6. Visualizing the results

The Gatling [25] plugin for Jenkins is able to archive Gatling reports that are created during builds. The plugin can be activated for a build via a post-build action in the build configuration. Then, it scans the workspace directory for reports and archives them, after every build. Furthermore, it shows a trend report on the project page for the stress tests of previous builds, i.e. “1” in Figure 3.8 and provides a link to further statistics as well as the particular HTML reports, i.e. “2” in Figure 3.8.

As stated in Section 3.1, the results of the time series analysis are provided by FPB in the form of CSV files. A Python script executes FPB analyzes during the build and generates an HTML report that contains a visual representation of the results. This
3. Implementation of a Continuous Integration toolchain

The script expects the FPB jar in the specific location `/vagrant/fpb/fpb.jar` and supports two commandline options. First of all, the option `-f <filename>` determines in which file and format the generated report is stored. Furthermore, a user can provide a threshold value via `-t <value>`, which the script uses to add color feedback to the result.

First, the script extracts the option values, if available, and executes FPB with the remaining arguments to start the analysis. Additionally, it retrieves the name of the analyzed metric and the analysis method from the FPB arguments. After the analysis completes, it processes the resulting CSV files that contain the result time series, as well as the input series.

Next, the data mining library `pandas` is used to process these time series, i.e. reading the CSVs and plotting the data. Though pandas is not actually a plotting library, it just wraps the standard Python plot library `matplotlib` in order to provide an easier method to create plots and improve them visually. The created plots consist of two subplots with the input series at the top and the result series at the bottom. Additionally, the subplots are titled with the metric name (top) and the analysis name, with "result" appended (bottom).

In case a threshold was specified, the script further colors both subplots using the following rules. Each point in the area under both time series is assigned a color in regard
3.6. Visualizing the results

Figure 3.9.: Visualization of an FPB analysis

to the result value $r$ at that x-coordinate and the threshold $t$. The point is colored green, if $r - t <= 1.0$, yellow, if $1.0 < r - t < 1.1$, and red, if $r - t >= 1.1$. The example in Figure 3.9 shows the resulting plot.

Finally, the script saves the plot in the target file, if it was specified, or tries to show it to the user as interactive version, otherwise. For files that end with “html”, the script will generate a PNG image and embed it, as base64 string, in an HTML document. For other files, the option is simply passed on to an API method that saves the plot.

In order to provide this visualization in the CI environment, this thesis uses the HTML Publisher. The plugin is easy to configure as a post-build action. It takes a directory and the name of an index page inside that directory. Since the HTML Publisher archives the whole directory, it is reasonable to create an empty directory during the build and save the HTML report of the Python script in it. The HTML Publisher will then archive the report after every build and provide a link to it on the project page.
4. Machine Learning

This chapter introduces machine learning and time series analysis methods. Section 4.1 covers machine learning, classification, and clustering and introduces two common algorithms in those fields. Afterwards, Section 4.2 describes selected techniques. A distance measure for time series is established, as well as a method to extract frequently occurring patterns and an averaging method. Finally, Section 4.3 introduces a new algorithm that attempts to detect anomalies in time series.

4.1. Machine Learning

*Machine learning* (ML), receiving increasing attention over the last years, is a discipline that emerged from the general field of *artificial intelligence*. The term was first defined by Arthur Samuel in 1959, as a “field of study that gives computers the ability to learn without being explicitly programmed” [61]. A more formal definition stems from Tom M. Mitchell [53]:

A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E.

Max Welling [82] circumscribes ML, in regards to experience, further. Giving the gist of his words, experience is *not remembering and regurgitating* observed information, but *transferring or generalizing* properties of observed data onto new, yet unobserved data.

In summary, ML enables a machine to perform a task, with increasing performance, as it gathers experience. The machine gathers experience by generalizing specific properties of observed information, and utilizing those properties on new, unobserved data. Before going into more detail, some terms are introduced.

**Definition 4.1.1.**  *Example:* An instance of data, e.g. a picture. It is sometimes represented as a *feature vector*.

**Definition 4.1.2.**  *Training data:* Examples used to train a program. Are either *labeled*, or *unlabeled*. 
4. Machine Learning

Definition 4.1.3. Query data: Examples unknown to a program. A program is supposed to utilize experience from the training data, to perform a task on query data. Query data is also provided to measure the performance of a program.

Definition 4.1.4. Features: A set of properties associated to an example, e.g. pixels in a picture.

Definition 4.1.5. Labels: Values or categories assigned to examples, e.g. “picture shows lion”.

A crucial task here is separating the defining, generalizable features of examples, from the non-generalizable. The non-generalizable features are called noise. If too many specific features of the training data are applied to query data, this is called over-fitting. Imagine a ML program, designed to identify lions. This program analyzed a multitude of pictures, showing lions, but also noise, in the form of, surroundings. One picture may show a lion in a savanna, while another may show one in a zoo. The over-fitting program does not realize that both pictures show a lion.

However interpreting too many features as noise is not reasonable either. This is called over-generalizing.

Consider another ML program, also to identify lions. The program, training on the same pictures, may consider having fur, as the only defining feature of a lion. When the over-generalizing program is presented with a picture of a squirrel, it identifies it as a lion.

ML is therefore about separating noise from defining features. The explanation and examples are based on Welling’s book [82].

Although there are many fields of ML, most common learning scenarios can be categorized as supervised learning, unsupervised learning, and semi-supervised learning. Foundations of Machine Learning [56] gives further insight on these approaches:

Supervised learning A program receives labeled training data. Its task is to assign query data to labels. Common applications include Classification and Regression.

Unsupervised learning A program receives unlabeled training data. The training data is also the query data. Its task is to predict, arrange or categorize information in some way. However, the program accomplishes this task without outside assistance. Clustering and Dimensionality Reduction are popular applications of supervised learning.

Semi-supervised learning A program receives both labeled and unlabeled training data. Its task is to label the query data. The general idea is to aid supervised learning, using labeled examples, by including the distribution of unlabeled examples into the learning process. Semi-supervised learning is often used when labeling examples is expensive, but acquiring unlabeled ones is cheap. For instance, oil drilling is such an
4.1. Machine Learning

The training data of KNN consists of examples in the form of tuples \((X_n, Y_n)\). \(X_n\) denotes the features of example \(n\), while \(Y_n\) denotes the label assigned to \(X_n\). \(Z\) denotes the query data. Figure 4.1 shows the algorithm in pseudo code. The function \(\text{dist}\) is called a distance measure and depends on the data. Additionally, the chosen distance measure is often critical for the performance of the algorithm. A popular distance measure is the Euclidean distance.

---

4.1.1. Classification

The aim of classification is to assign examples of the query data to specific categories, represented by labels. For instance, in cancer research, classification could be used to determine the risk of a patient to become diseased with cancer. Training data consists of characteristics of cancer patients and other patients. This data is labeled whether the patient has cancer, or not. A classification algorithm can then determine what “features” of a patient possibly lead to his disease. Afterwards, it can classify if new patients are either under a high, or a low risk to disease of cancer.

Although classification and regression are related applications, the distinction lies in the output. While classification identifies some group membership for the query data, regression tries to predict a continuous variable, e.g. stock price, based on the training data.

K-Nearest-Neighbors

K-Nearest-Neighbors (KNN) is a typical algorithm used in classification. It is perhaps one of the simplest machine learning algorithms. KNN can be used for regression as well. The following introduces the classifier version of KNN, based on \(\text{dist}\). Afterwards it explains the difference to the regression version.

The training data of KNN consists of examples in the form of tuples \((X_n, Y_n)\). \(X_n\) denotes the features of example \(n\), while \(Y_n\) denotes the label assigned to \(X_n\). \(Z\) denotes the query data. Figure 4.1 shows the algorithm in pseudo code. The function \(\text{dist}\) is called a distance measure and depends on the data. Additionally, the chosen distance measure is often critical for the performance of the algorithm. A popular distance measure is the Euclidean distance.
Euclidean Distance (ED), defined as:

$$\text{dist}(p, q) = \sqrt{\sum_{i=1}^{n} (q_i - p_i)^2}$$

Variations of the algorithm include weighting the neighbors based on the distance, such that nearer neighbors are deemed more important. Furthermore, KNN can also be used for regression. Instead of choosing the most used label across the nearest neighbors, calculate the (weighted) mean of the labels of all nearest neighbors and assign it to the example.

4.1.2. Clustering

Clustering or cluster analysis is the process of partitioning query data into homogeneous regions, as formulated by Foundations of Machine Learning [56]. Those homogeneous regions are called cluster. What a cluster is, in detail, depends on the approach and the algorithm itself.

For instance, hierarchical clustering builds a tree of clusters, where each cluster except the root has a parent cluster, of which it is an element. The root contains all clusters. Hierarchical clustering is either agglomerative, or divisive. Agglomerative hierarchical clustering is the “bottom up” approach. Each example starts in its own cluster. In each step, clusters are merged, until only the root cluster remains. Divisive hierarchical clustering on the other hand is the “top down” approach. All examples start in the root cluster. In the process, clusters are split recursively, until a break condition is satisfied.

As an example, the problem of color quantization is a good application for clustering. Color quantization aims to map an image of a high color density, to the closest possible representation of the same image, in a lower color density. In Foundations of Machine Learning [56], this problem is solved with the k-means algorithm.

K-means

Figure 4.2 shows k-means in pseudo code. A cluster, in k-means, consists of a centroid and one or multiple examples. The centroid changes over time and is no example itself, but rather the mean value of all examples in the cluster. The general idea is to assign every example of the training data to the cluster, with the nearest centroid. To determine the nearest centroid, again a distance measure dist is used. After a cluster changes, the algorithm needs to update the respective centroids. It repeats those steps until it converges, meaning the clusters do no longer change after an iteration. Other termination conditions, e.g. the number of iterations reached a threshold, are possible.
4.2. Time series analysis

This section introduces selected methods for time series analysis. These methods will be revisited, and used as subroutines in the algorithm, proposed in Section 4.3. The literature [54] defines the term time series as follows:

A time series represents ordered real-valued measurements at regular temporal intervals.

In [52], the terms time series, and subsequence are formally defined as follows:

**Definition 4.2.1.** Time Series: A time series $T = t_j, \ldots, t_m$ is an ordered set of $m$ real-valued variables.

**Definition 4.2.2.** Subsequence: Given a time series $T$ of length $m$, a subsequence $C$ of $T$ is a sampling of length $n < m$ of contiguous position from $T$, that is, $C = t_p, \ldots, t_{p+n}$ for $1 \leq p \leq m - n + 1$.

Before they can be analyzed, time series need to be normalized. A popular normalization method is z-normalization [69].

**Definition 4.2.3.** Z-normalization: Given a time series $T$, the time series

$$T_Z = \frac{(T - mean(T))}{std(T)},$$

is normalized.
with a mean of zero, and a standard deviation of one, is called z-normalized.

Normalization is necessary, as it was empirically proven \[43\], and reported that comparing non-normalized time series is meaningless \[69\] \[45\].

### 4.2.1. Dynamic Time Warping

Dynamic Time Warping (DTW) is a distance measure to determine similarity between two time series. According to Stan and Chan \[73\], it was introduced in 1983 \[47\]. A popular application of DTW is in speech recognition. It is used to recognize spoken words by different voices. The following description is based on \[73\].

Euclidean distance (ED), introduced in Section \[4.1.1\], is already an efficient distance measure, that can be adjusted to work with time series. Given two time series \(X\), and \(Y\), of length \(|X|\), and \(|Y|\), respectively,

\[
X = x_1, x_2, \ldots, x_i, \ldots, x_{|X|} \\
Y = y_1, y_2, \ldots, y_i, \ldots, y_{|Y|}
\]

the ED \(dist_{ED}\) between \(X\), and \(Y\), is defined as:

\[
dist_{ED}(X, Y) = \min(|X|, |Y|) \sum_{i=1}^{\min(|X|, |Y|)} \sqrt{(x_i - y_i)^2}
\]

Although ED is efficient, it has a major drawback, in that it does not consider time shifts. For instance, if two time series are identical, but one is shifted slightly, in time, ED might determine a high distance between these, although they are almost identical. In addition, ED cannot handle time series of different length. If one series is longer, than the other, ED is only calculated over the length of the shorter series.

The motivation for DTW was to provide a more intuitive distance measurement for time series that ignores time shifts in the data.

**Problem**

Given two time series \(X\), and \(Y\), defined as before, construct a warp path \(W\)

\[ W = w_1, w_2, \ldots, w_K \quad \max(|X|, |Y|) \leq K < |X| + |Y| , \]

where each element of the warp path is a tuple \((i, j)\) of indices, referring to points \(X_i\), and \(Y_j\), in the time series. A warp path has to fulfill certain requirements to qualify as such. First, the warp path must start at the first element of each time series and end at the last, therefore:

\[ w_1 = (1, 1) \quad w_K = (|X|, |Y|) . \]
4.2. Time series analysis

```python
def dtw(X, Y):
    # Initialize matrix of size ||X||,||Y||
    costmatrix = matrix([|X|, |Y|])
    for i in [1,|X|]:
        for j in [1,|Y|]:
            # retrieve the lowest cost from adjacent fields, if possible
            index, cost = minCost(i, k, costmatrix)
            # chose the minimum available distance
            costmatrix[i][j] = cost + dist(X[i], Y[j])

    warppath = []
    position = ([|X|],|Y|)

    while position is not (1,1):
        warppath.append(position)
        position, cost = minCost(position, costmatrix)
```

Figure 4.3.: Pseudo code for DTW

Second, the indices have to occur monotonically increasing, in the warp path. Third, every index of each time series has to occur at least once:

\[ w_k = (i, j), w_{k+1} = (i', j') \quad i \leq i' \leq i + 1, j \leq j' \leq j + 1. \]

An optimal warp path has a minimum distance, where the distance of a warp path \( W \) is

\[ dist(W) = \sum_{k=1}^{K} dist(w_{k_i}, w_{k_j}). \]

\( dist(w_{k_i}, w_{k_j}) \) is some distance measure between the data points \( w_{k_i} \), and \( w_{k_j} \), in the \( k^{th} \) element of the warp path. Usually, the distance measure is ED.

**Dynamic Programming Approach**

Generally, the optimal warp path is calculated, using dynamic programming. Figure 4.3 shows this approach in pseudo code. The algorithm solves the problem for a larger time series, by solving it for subsequences of the time series and using the solutions to gradually solve it for larger time series, until a solution for the entire series is found. It achieves this in two steps. First, it constructs a cost matrix \( C \) of size \(|X| \times |Y|\). The rows, and columns of this matrix represent points in the time series \( X \), and \( Y \), respectively. Therefore, each coordinate in the cost matrix represents a link between two points of both series. The distance between two points, \( x_i \) and \( y_j \), is interpreted as a cost \( cost_{ij} = dist(x_i, y_j) \). The algorithm starts in the top left corner \( C_{00} \), and fills the entry with the distance between the first two points, \( x_0 \) and \( y_0 \). The algorithm iterates row wise over the \( C \), filling each entry \( C_{ij} \) with the sum of the minimum cost of the upper left, left, or top neighbor, and the cost of the entry itself, such that

\[ C_{ij} = \min(C_{i-1j-1}, C_{ij-1}, C_{i-1j}) + cost_{ij}. \]
4. Machine Learning

$C_{0,0}$ contains the optimum warp path between $x_0$ and $y_0$. Since every step only adds the minimum possible cost to the warp path, the entry $C_{i,j}$, contains the total cost of the optimum warp path between

\[
X' = x_1, \ldots, x_i \quad \text{and} \quad Y' = y_1, \ldots, y_j.
\]

Therefore, $C_{|X|,|Y|}$ contains the cost of the optimum warp path between $X$ and $Y$. After it constructed the matrix, the algorithm extracts the optimal warp path via backtracking. Beginning in the bottom right corner, the adjacent field, with the lowest cost, upper left, left, or top of the current element, is the next element in the optimum warp path. When the backtracking reaches the top left field, the warp path is complete.

Since every entry of the cost matrix is filled exactly once and this happens in constant time, the complexity of the dynamic programming approach has an upper bound of $\mathcal{O}(|X| \cdot |Y|) = \mathcal{O}(N^2)$.

**FastDTW**

Salvador and Chan [73] proposed a faster implementation of DTW, called *FastDTW*. FastDTW uses a three-part approach to speed up the regular implementation:

- **Coarsening**: Shrink a time series into a smaller time series, that represents the same curve as accurately as possible
- **Projection**: Find an optimum warp path for a lower resolution and use it as a basis for an optimum warp path on a higher resolution
- **Refinement**: Refine the basis warp path, to have a lower distance in the higher resolution

Figure 4.4 illustrates how the algorithm works. In the coarsening step, FastDTW splits the length of both time series in half, until a base case is reached. Adjacent points of each time series are averaged, to create a time series of half the size of the original series. When the base case is reached, the standard DTW implementation, described in the previous section, is called on the lowest resolution. The base case is represented by the left most square in Figure 4.4. Since every coarsening step halves the length of each time series, each lower resolution point maps to at least four higher resolution points, the dark shaded squares. The projection step constructs a search window from this warp path. This search window contains the points in the warp path and a number of adjacent cells, represented by the light shaded squares, around the warp path. The *radius* parameter determines how many adjacent cells are added to the search window in each step, i.e. in Figure 4.4 radius is set to 1. Finally, the refinement step calls a constrained version of DTW, with the search window as argument. This version only evaluates the cells in the
4.2. Time series analysis

Figure 4.4.: How FastDTW evaluates the warp path on different resolutions, with $\text{radius} = 1$. Taken from [73]

search window, instead of the whole cost matrix. The final result is shown in the rightmost square in Figure 4.4.

Figure 4.5 shows the recursive implementation of FastDTW. The base case is determined by the radius parameter. Since the length is halved in each step, the minimum meaningful length of a time series is $\text{radius} + 2$. If a time series has length $\text{radius} + 2$, or smaller, the warp path would be evaluated twice, since the constrained DTW would calculate every point in the lower resolution warp path again. For instance, in Figure 4.4 the $1/8$ resolution search window has sides of length $2 \leq \text{radius} + 2$, since radius is 1. If the algorithm would halve this window again, it would evaluate a warp path for the bottom left and the top right square of the $1/8$ resolution. However, when it would refine this warp path in the $1/8$ resolution, it would still calculate a warp path inside the same light shaded squares. Therefore, it would evaluate the same warp path twice.

Until the implementation reaches the base case, it reduces each time series by half. Then, it finds a lower resolution warp path recursively. For each warp path, it refines the path, using the constrained DTW, with a search window, based on the lower resolution path and $\text{radius}$.

According to [73], FastDTW has an upper bound complexity of $O(N)$, because the warp path grows only linear, instead of quadratic, with the time series. This is a large improvement, over the $O(N^2)$ upper bound, of the standard approach.

4.2.2. Time series motifs

The term motif originated in computation biology [77]. It was transferred to time series analysis [52], as a previously unknown, frequently occurring pattern in time series data.
4. Machine Learning

```python
def FastDTW(X, Y, radius):
    minTSsize = radius + 2
    if (|X| ≤ minTSsize or |Y| ≤ minTSsize):
        # call standard DTW
        return DTW(X, Y)
    else:
        shrunkX = X.reduceByHalf()
        shrunkY = Y.reduceByHalf()
        # call FastDTW recursively
        lowResPath = FastDTW(shrunkX, shrunkY, radius)
        # calculate a higher resolution search window from the lower resolution warp
        window = ExpandedResWindow(lowResPath, X, Y, radius)
        # call constrained DTW
        return DTW(X, Y, window)
```

Figure 4.5.: Pseudo code for FastDTW, based on [73]

Problem

Before defining the motif problem, some other definitions are necessary. The following definitions are taken from [52].

**Definition 4.2.4.** *Match:* Given a positive real number \( R \) (called *range*) and a time series \( T \) containing a subsequence \( C \) beginning at position \( p \) and a subsequence \( M \) beginning at \( q \), if \( D(C, M) ≤ R \), then \( M \) is called a matching subsequence of \( C \).

A match is therefore a subsequence that features a distance equal, or lower, than \( R \) to at least one other subsequence in a time series. Whereat the distance is determined by some distance measure \( D \), e.g. DTW. To prevent an inflation of matches, the term *trivial match* is introduced.

**Definition 4.2.5.** *Trivial Match:* Given a time series \( T \), containing a subsequence \( C \) beginning at position \( p \) and a matching subsequence \( M \) beginning at \( q \), \( M \) is called a trivial match to \( C \) if either \( p = q \) or there does not exist a subsequence \( M' \) beginning at \( q' \) such that \( D(C, M') > R \), and either \( q < q' < p \) or \( p < q' < q \).

This means that a match is considered trivial, if the subsequence \( C \) is directly next to the matching subsequence \( M \). For instance, when sliding a window over the time series \( T \) that starts at \( q \) and ends at \( p \), consider the resulting subsequences \( M' \). \( C \) is considered a trivial match to \( M \), when all of these subsequences \( M' \) are a match to \( C \). In other words, a non trivial match requires that at least one non match exists between the two subsequences. Now the *K-Motifs* problem can be defined.

**Definition 4.2.6.** *K-Motifs:* Given a time series \( T \), a subsequence length \( n \) and a range \( R \), the most significant motif in \( T \) is the subsequence \( C_1 \) that has the highest count of non-trivial matches. The \( K^{th} \) most significant motif in \( T \) is the subsequence \( C_K \) that has the \( K^{th} \) highest count of non-trivial matches.
4.2. Time series analysis

Note that this definition slightly differs, in two aspects, from the definition in \cite{52}. First, \cite{52} breaks ties by choosing the motif with the lower variance. In this thesis, ties are not broken for simplicity. Second, the publication \cite{52} demands the motifs to be mutually exclusive. Since the method introduced here, finds motifs using a symbolic representation of relatively small words, a mutually exclusive definition would not make sense.

Next, a method to solve the K-Motifs problem is introduced. For ease of reference, this method will be called \textit{SAXMotifs}, although the author \cite{74} did not provide a name. First, \textit{SAXMotifs} splits a time series into equally sized pieces and builds a symbolic representation of each piece. Next, the symbolized pieces and the frequency of their occurrence are stored in a data structure. Finally, the algorithm retrieves all pieces, sorts them ascending, depending on their frequency, and does some post processing. Then, the last element is the most significant motif, and the last \( K \) elements are the K-Motifs.

\textbf{SAX}

\textit{SAXMotifs} uses \textit{SAX} (\textbf{S}ymbolic \textbf{A}ggregate \textbf{A}proximation), as symbolic representation for a time series. \textit{SAX} was first introduced in \cite{52}. It was later explained and evaluated more detailed in a stand-alone publication \cite{51}. The following description of \textit{SAX} is based on \cite{51}.

\textit{SAX} reduces a time series of arbitrary length \( n \) to a string of arbitrary length \( w \), with \( w < n \). In general, \( w \) is chosen much smaller than \( n \). The resulting string consists of characters from an alphabet, with alphabet size \( a \), \( a > 2 \), e.g. \{“\( a'’ \), “\( b'’ \), “\( c'’ \)\} for \( a = 3 \).

Before \textit{SAX} is applied, the time series is transformed to a \textit{z-normalized} form. As stated in Section 4.2, it is meaningless to compare time series that are not \textit{z-normalized}. \textit{SAX} works in two steps. First, it applies the dimensionality reduction method \textit{Piecewise Aggregate Approximation (PAA)} to a time series. Second, it symbolizes the approximated representation, to create a discrete representation of the original series.

\textit{PAA} reduces a time series \( C \) of length \( n \) to an approximated time series \( \bar{C} \) of length \( w \). \( C \) is split into \( w \) equal sized \textit{frames} as illustrated in Figure 4.6(a). The mean value of all points in \( C \) falling into a frame represents a data point in \( C \). Formally, each element \( \bar{c}_i \) of \( \bar{C} \) is calculated with the following formula:

\[
\bar{c}_i = \frac{w}{n} \sum_{j=\frac{n}{w}(i-1)+1}^{\frac{n}{w}i} c_j .
\]

The \( i^{th} \) frame includes the \( \frac{n}{w} \) data points from index \( i - 1 * \frac{n}{w} + 1 \) to index \( \frac{n}{w} * i \). The values are added together and their mean is calculated, by multiplying with \( \frac{1}{\frac{n}{w}} \), or \( \frac{w}{n} \). If \( n \) is not a multiple of \( w \), the last, incomplete frame is omitted.
4. Machine Learning

(a) PAA representation of a time series

(b) Symbolic representation with \( a = 3 \)

(c) Lookup table for \( 3 \leq a \leq 5 \)

Figure 4.6.: Shows the PAA (a) and the symbolic representation (b) (both taken from [51]) of a time series, and the lookup table (c) (taken from [45]).

After the PAA is available, SAX can create a discrete, symbolic representation of the time series. Figure 4.6(b) shows the symbolic assignment, for \( a = 3 \). SAX produces discrete symbols with equal probability. This is achieved since a z-normalized time series has a Gaussian distribution. The authors of [51] refer to [48] for this assertion. Since the density of the values in the time series is therefore determined by a bell curve, SAX can define breakpoints to split the area under the curve into equally sized parts. To produce \( a \) equally sized parts, \( a - 1 \) breakpoints \( \beta \) are needed. These breakpoints can be calculated in a statistical table beforehand, and simply looked up when needed. Figure 4.6(c) shows such a table for \( 3 < a < 5 \), while the breakpoints are represented by the two (since \( a = 3 \)) horizontal lines in 4.6(b). For instance, all elements \( \bar{c}_i \) of the PAA that are below the smallest breakpoint \( \beta_1 \) (\( \bar{c}_i < \beta_1 \)) are mapped to “a”, all elements \( \bar{c}_j \) that are greater or equal to \( \beta_1 \), and lower than the second smallest breakpoint (\( \beta_1 \leq \bar{c}_j \leq \beta_2 \)) are mapped to “b”, etc. Figure 4.6(b) illustrates the resulting assignment for each point in the PAA. The result of this time series discretization is called a word, e.g. the time series in Figure 4.6(b) is represented by the word \texttt{baabccbc}. According to the authors [45], an alphabet size of \( a = 3 \), was empirically proven to be the best choice for virtually any dataset and task.

SAX trie

A trie [22] is a tree data structure, which can store, and retrieve a large amount of keys rather cheap. Typically, a key is represented by a string, though it could be any kind of
ordered structure, e.g. numbers. For brevity, only string tries are considered. A string is retrieved from the trie, by removing a prefix, i.e. the first character, and looking for a branch of the current node that is labeled with that character. This process starts at the root node, and is continued, until either a leaf node is reached, or the desired branch does not exist. If a leaf node is reached, the string exists in the trie. However, if the search stops in an inner node, the string is not stored. Generally, a trie does not require leaf nodes to store any information, apart from the fact that they are leaf nodes.

\cite{context} uses such a trie structure in combination with an array to store, and retrieve time series in SAX representation. This thesis refers to the structure, consisting of the trie, and the array, as SAX trie. The algorithm begins, by sliding a window of length $n$ across a time series and extracting subsequences for each window. Again, the last window is omitted, if the length of the time series is not a multiple of $n$ as before with PAA. The extracted subsequences are then converted to SAX words of size $w$ with alphabet size $a$. Afterwards, each word is stored in an array, such that the index corresponds to the index in the original sequence. After the array construction, the algorithm builds a trie from the array, inserting each word from the array. This trie stores all indices of occurrences of the word $w$ in the array in a linked list in the leaf that corresponds to $w$. Additionally, the number of occurrences can be mapped back into an additional column in the array. This provides a fast method to pick an arbitrary index in the original time series and find similar occurrences of the subsequence, beginning at that index. Furthermore, both structures can be built in linear time and space, and take up significantly smaller space than the original time series, since each SAX symbol can be stored in $|\log_2(a)|$ bits.

**SAXMotifs**

\textit{Jmotif} \cite{74} implements \textit{SAXMotifs}, among other algorithms. Unfortunately, there is no publication about this implementation. Therefore, this description is based on source code from \cite{74}.

\textit{SAXMotifs} uses the SAX trie structure to find $m$ motifs of length $n$. A major difference to the original approach is that the \textit{SAXMotifs} trie does not differ between the word length and the sliding window length, both values are set to $n$. This effectively makes every word in the trie a potential motif. Therefore, the algorithm only needs to select the most promising motifs from the trie, instead of finding them separately. This leads to a very fast, and space efficient computation.

\textit{SAXMotifs} begins by sliding a window over the time series, and building the SAX trie, as described in the previous section. After the initialization, the structure can detect motifs. The algorithm retrieves a list of all words from the array and sorts them in ascending order, depending on their frequency. Then, it iterates backwards over the list, since the words with the highest frequencies are the most promising. As soon, as
it extracted $m$ motifs, the calculation is abandoned. Each word is then checked for three fail conditions. Is the word a motif? In order to qualify as a motif, a word must occur at least twice. Was the word already processed? Since words can, and should, occur multiple times in the array, SAXMotifs keeps a set of already seen words to avoid repeated processing of the same word. Is the word trivial? A non-trivial word consists of at least two different characters. Thus, simple words that represent linear sections, due to containing only a single character, do not have a distractingly high number of occurrences in the windows left and right of them. Each word that passes these checks is considered a motif. SAXMotifs constructs a motif record for each motif, containing the word, and all indices of its occurrence in the original time series. Finally, the algorithm returns a list of these motif records after it found $m$ motifs, or there are no words left.

### 4.2.3. Averaging of time series

Time series averaging is important, since many distance-based algorithms require, in addition to a distance measure, an average measure, e.g. k-means (introduced in Section 4.1.2). When averaging time series, it is important to consider time shifts. Therefore, DTW seems suitable to construct an averaging method for time series, but according to [67], past attempts provide either an inaccurate average, or even break the convergence properties of algorithms using them. As a result, they proposed a new method, named dtw barycenter averaging (DBA).

DBA takes an initial average sequence and a list of time series, and constructs a new average sequence. Figure 4.7 shows the algorithm in pseudo code. DBA calculates DTW alignments for the given average sequence and each time series. These alignments are then added to an array of sets. Such that each index in the array corresponds to an index in the average sequence, while the set at this index contains values of the points aligned with this index. Consequently, a single point in a sequence can influence multiple points.

```python
def DBA(center, sequences):
    alignments = array(len(center))
    newCenter = []
    for seq in sequences:
        # Get the warp path consisting of tuples (i, j) with i ∈ center ∧ j ∈ seq
        path = DTW(center, seq)
        # Iterate over the warp path from (len(center), len(seq)) to (1, 1)
        for i, j in reverse(path):
            alignments[i] = alignments[i] ∪ seq[j]

    for alignment in alignments:
        # The arithmetic mean of the alignment becomes a data point in the new center
        newCenter.append(mean(alignment))
```

Figure 4.7.: Pseudo code for DBA
4.3. ATC

This Section introduces a new classification algorithm, called Automatic Time Series Classification (ATC). When analyzing run-time metrics of software, the aim is usually to detect aberrant behavior, e.g. memory leaks. This is called anomaly detection. Time series of run-time metrics, e.g. memory usage, show different patterns when an anomaly is present. Human experts can identify these patterns and therefore isolate the time frame in which the software behaved unusual. However, this is temporally expensive, especially with larger volumes of data. ATC combines multiple, well researched methods, in order to provide an automated form of such anomaly detection to assist a human expert.

4.3.1. Idea

An intuitive approach to assess behavior of a software is to learn from previous behavior. However, aberrant behavior occurs rarely by definition. Therefore, aberrant behavior is hard to learn, since there are less examples, or none at all, for it in previous behavior. In contrast, normal behavior occurs frequently, since it is the absence of aberrant behavior. Consequently, normal behavior is easy to learn, since there are many examples for it. Hence, an algorithm that learns to detect normal behavior seems more promising to succeed than one that learns to detect aberrant behavior. Nevertheless, an algorithm that is able to detect normal behavior can detect aberrant behavior, too, by the exclusion principle: Everything that is not normal behavior, therefore, must be aberrant behavior. Thus, the algorithm needs a model, an approximation, of normal behavior, which it can apply to new behavior in order to classify it as aberrant, or normal.

The behavior of software is represented by time series of its run-time metrics. When the software shows aberrant behavior, a part of the time series contains an anomaly. In this scenario, an anomaly is every part of the data where the algorithm cannot apply the model. It is reasonable to separate the algorithm in two steps. First, a training step, where the algorithm uses previously collected time series, to build a model for normal behavior. Second, a query step, where the algorithm applies the model to new data and classifies it.

How to construct such a model for normal behavior? The idea of the training step is to find the most characteristic patterns in the time series and use them as a “dictio-
4. Machine Learning

The algorithm extracts patterns from the time series that have a high number of similar counterparts and forms groups out of the most similar. Each group has a center, representing the average of its elements. An element is assigned to the group, to whose center it has the lowest distance. The algorithm tries to find the most characteristic grouping such that the elements in a group are as similar as possible, while the groups among each other are as dissimilar as possible. In other words, the algorithm tries to maximize the distance between the groups and minimize the standard deviation in each group. In order to find a more characteristic grouping, the algorithm removes each group, whose standard deviation exceeds the mean distance of that group’s center to all other centers. From there, the patterns are again assigned to their nearest groups. This process is repeated until the algorithm converges, i.e. the grouping does no longer change. Afterwards, the centers of all groups form a dictionary of normal behavior. Each time series that is considered normal, within the scope of the algorithm, can be approximated by the patterns in the dictionary.

Since anomalies generally occur infrequent, it is not reasonable for the query step to classify a large time series completely as aberrant or normal. Better than applying the model to a time series, as a whole, is to apply it to smaller parts separately. This also assists an expert in narrowing the time frame of aberrant behavior down. Each part of the series is compared to all patterns of the model. If the distance to the nearest pattern does not exceed a threshold value, the part is classified as normal. Otherwise, the part is considered an anomaly. Though which value is the best threshold? The threshold introduces a tradeoff. If it is set too low, this may lead to normal behavior being considered an anomaly. This is called a false positive, since the algorithm thinks it found an anomaly, but actually found normal behavior. Conversely, if the threshold is set too high, an anomaly may go undetected. This is called a false negative, since the algorithm incorrectly classifies an anomaly as normal behavior. The best threshold minimizes false positives and false negatives.

4.3.2. Algorithm

To find frequently occurring patterns, ATC uses the SAXMotifs algorithm, described in Section 4.2.2. This algorithm uses SAX, a symbolic representation for time series described in the same section, to find motifs and their occurrences. Next, it groups the extracted occurrences into clusters, using a modified k-means algorithm. K-means, introduced in Section 4.1.2, requires a distance as well as an average measure. For its ability to recognize time shifts, DTW, introduced in Section 4.2.1, was chosen as a distance measure. ATC uses the DTW-based average measure DBA, introduced in Section 4.2.3, for the same reason. Since k-means needs to use the distance measure rather often, ATC uses FastDTW, also introduced in Section 4.2.1, to calculate the DTW distance in linear, instead of quadratic time. As radius for FastDTW it uses 30, as that yields the most accurate optimal warp path, while still reducing the computing
time, according to [73]. Note that ATC assumes all data points are measured in equal
time intervals. Therefore results for time series featuring different time intervals are not
reliable. As stated in the previous section, ATC consists of a training and a query step,
which are named ATCTrain and ATCQuery respectively.

**ATCTrain**

Figure 4.8 shows the pseudo code for ATCTrain. It takes a time series as input and pro-
duces a dictionary of equally sized, characteristic patterns of that series. First, it calls
the SAXMotifs algorithm to receive a list of motif records of length patternLength. The
alphabet size controls how much the SAX representation abstracts the time series. Since
the data will be reduced later, the initial data should be as comprehensive as possible.
Therefore, SAXMotifs should find as many motifs as possible. Since each motif consists
of at least one data point, and needs at least one non-trivial match, half the length of
the series is an upper bound for the number of motifs. As described in Section 4.2.2
SAXMotifs returns motif records that contain starting positions of all occurrences of a
pattern. Nevertheless, ATCTrain needs to extract these occurrences from the original
time series by extracting the subsequence of length motifLength, beginning at the start-
ing position of the occurrence. Furthermore, it removes trivial matches that cannot be
detected when using the SAX representation. These are simply occurrences that start
only one index apart from each other. Additionally, it adds the first occurrence of each
motif record to a list of examples in order to provide an initial clustering later. Finally,
it starts the clustering algorithm. After the algorithm returns, the centers of the clusters
form the desired dictionary. The extraction of these centers is omitted in Figure 4.8 for
brevity.
4. Machine Learning

```python
def cluster(series, examples, iterations):
    # Initialize len(examples) cluster with the examples as center
    clusterList = InitializeCluster(examples)

    iter = 0, converged = False
    while not converged and iter < iterations:
        for s in series:
            minDist = Inf, nearest = None
            for c in clusterList:
                dist = FastDTW(s, c.center, 30)
                if dist < minDist:
                    minDist = dist
                    nearest = c
                    nearest.add(s)

        for c in clusterList:
            if c.isEmpty():
                clusterList.remove(c)
            else:
                c.updateCenter()

        toRemove = []
        for c in clusterList:
            mean = meanDistToAll(clusterList)
            if mean < c.stdDev:
                toRemove.append(c)
        clusterList.removeAll(toRemove)
```

Figure 4.9.: Clustering algorithm of ATCTrain

Figure 4.9 shows the k-means based clustering algorithm. Typically, k-means initializes the clusters by choosing $k$ random elements from the data set. However, since the data set, in this case, consists of multiple groups of similar elements, this approach is not reasonable. Instead, the algorithm initializes the clusters with the previously collected examples as centers. This prevents a clustering that consists only of multiple occurrences of the same pattern, which would produce very similar clusters, with very dissimilar elements in them. Recalling from the previous section, the aim was to produce the opposite: Dissimilar clusters with similar elements.

After the clusters are initialized, the algorithm gradually improves the clustering, until it converges, or reaches a certain number of iterations. Each iteration begins with assigning all series to their nearest cluster under the DTW distance, which is calculated efficiently using FastDTW. Next, the clusters are updated by the subroutine `updateCenter`. If a cluster contains no elements, it is not updated, but removed.

Figure 4.10 shows the pseudo code for `updateCenter`. To receive the most accurate average over all elements in the cluster, DBA is used. As stated in Section 4.2.3, DBA is an iterative, converging algorithm. Therefore the process is repeated, until the computed average of iteration $i$, does no longer differ from the average of iteration $i - 1$. FastDTW is used again, to compare two averages. After DBA converges, the new center
def updateCenter()
  while True:
    newCenter = copy(self.center)
    DBA(self.center, self.elements)
    dist = FastDTW(self.center, newCenter)
    self.center = newCenter
    if dist == 0:
      break
  self.stdDev = calculateStdDev()

Figure 4.10.: Algorithm to update the center of a cluster

of the cluster is the computed average of the last iteration.
Finally, updateCenter also calculates the standard deviation of the cluster. The standard deviation $\sigma_C$ of a cluster $C$, with center $c$ can be defined by the following formula:

$$\sigma_C = \sqrt{\sum_{s \in C} (\mu_C - DTW(s, c))^2 / |C|}, \text{ with } \mu_C = \frac{\sum_{s \in C} DTW(s, c)}{|C|}.$$ 

Thereby is $\mu$ the mean distance from each element of a cluster to its center. The formula above just redefines the regular standard deviation under DTW distance.

Once, all clusters are updated, the algorithm tries to reduce the number of clusters. After the previous considerations, ATCTrain should remove each cluster, whose standard deviation exceeds its mean distance to all other clusters. This is done, by iterating over the clusters, and calculating this mean distance for each cluster. Each cluster that has a lower mean distance than its standard deviation, is booked for removal. Afterwards, all booked clusters are removed. Although the algorithm could also remove the offending clusters during the iterations, this would yield an unstable result. If a cluster is removed that affects the mean distances of following clusters, therefore different arrangements of the clusters would lead to different clusters being removed.

**ATCQuery**

ATCQuery, shown in Figure 4.11 calculates a time series of ratios between a threshold value, and the distance to the best matching pattern from a dictionary. It slides a window of a certain length over the query series, to create subsequences. Each subsequence is z-normalized, in order to be able to compare it to the patterns in the dictionary. Thereafter, it finds the nearest pattern under the DTW distance, using FastDTW. After this pattern is found, the result, in the range of the window, is set to the ratio of the distance to this pattern, and the threshold value. Note that the ratio of a point can change with multiple iterations, because the windows overlap each other, and each window overwrites the ratios of previous overlapping windows. If the distance to the nearest pattern was lower, or equal to the threshold, the ratio is lower, or equal to one.
4. Machine Learning

```python
def ATCQuery(dictionary, query, windowLength, threshold):
    i = 0, result = array(len(query))
    while (i + windowLength) < len(query):
        window = zNormalize(query[i:i+windowLength])
        minDist = Inf, nearest = None
        for pattern in dictionary:
            dist = FastDTW(window, pattern, 30)
            if dist < minDist:
                minDist = dist
                nearest = pattern
        result[i:i+windowLength] = minDist / threshold
        i = i + 1
    return result
```

Figure 4.11.: ATCQuery

Otherwise, it is greater than one. Since ATC is a classification algorithm, all points with a ratio greater than one can be interpreted as anomalous, and all other points as normal, in the sense of the algorithm.

4.3.3. Implementation

ATC is implemented as an analysis for Find Performance Bugs (FPB) [72]. FPB is an extensible analysis tool for time series, written in Java. It features a graphical user interface, as well as a CLI. Analyzes are either compiled directly into the FPB jar, provided by external jars, or by R programs. FPB provides a connection to a Solr server, where the time series are stored. The user chooses an analysis, and a time series in the Solr server, to analyze. Then, FPB retrieves the time series from Solr, and starts the chosen analysis. Afterwards, FPB generates CSV files for a result time series, and the original series.

ATC is implemented as two Java analyzes: ATCTrain, and ATCQuery. They are compiled into the FPB jar, and accessed via their package names. Since both ATCTrain, and ATCQuery expect arguments, but FPB does not support analyzes with arguments, this functionality was added.

Additional to the arguments alphabet size, and pattern length, specified in the previous section, ATCTrain expects a dictionary name. FPB provides the time series to analyze in form of a double array, containing the data points, and an array of Instants, containing the time stamps. First of all, ATCTrain takes the data points of the series to construct an ArrayTimeSeries, which is a wrapper class for the data points that inherits from a class in the Java-ML [36] library. This is necessary, since the implementation of FastDTW, used by ATCTrain, is part of this library, and expects objects of this class. Then, it calls the main algorithm, shown in Figure 4.8. For the SAXMotifs algorithm, a slightly modified version of the Jmotif implementation series2Motifs, is used. The
modifications are confined to reimplementation of some helper methods, e.g. PAA, or \textit{z-normalization}, in order to reduce the number of copy operations.

After the subsequences are extracted, the clustering algorithm, from Figure 4.9, is called. A cluster is represented by an instance of the class \textit{Cluster}. An instance features a list of its elements, its center, as well as methods to remove an element, add an element, update the center, or calculate the distance to another Cluster. A center is updated by the algorithm in Figure 4.10. The author [67] of the DBA algorithm provides a Java implementation on his website [21], which is used for this purpose. The distance to another cluster is calculated by executing FastDTW on the respective cluster centers. Since the clustering algorithm is computationally demanding, ATCTrain parallelizes the algorithm. This is done in three places, by dividing either the list of subsequence, or the clusters into equally sized chunks, and distributing them across multiple threads. First, when assigning the subsequences to clusters, the list of sequences is divided. Second, the clusters are distributed, and updated in parallel. Third, the reduction step at the end of each iteration, divides the list of clusters to determine which clusters should be removed.

Finally, ATCTrain extracts the centers of the clusters, to form a dictionary.

Instead of returning a result, it stores the produced dictionary in Solr, under the provided dictionary name. The Solr server uses the schema, described in Section 3.5. ATCTrain iterates over the patterns of the dictionary with a \textit{control variable}. The metric field is set to the dictionary name, while the measurement field is set to the current value of the control variable. All other fields are set to a constant string, to prevent the patterns from accidently appearing in other queries. The patterns receive dummy time stamps in an interval of one second, because they are later ignored anyway.

ATCQuery retrieves a set of dictionaries from the Solr server, and executes the algorithm in Figure 4.11. The names of these dictionaries are provided via the argument list. First, the dictionaries are retrieved from Solr, and each pattern is converted to an instance of ArrayTimeSeries. The actual algorithm from Figure 4.11 is again parallelized, to decrease the run time. This is achieved, by analyzing each window independently as a task, which can be completed by a thread. Each completed task produces a window result, containing the start, and end index of the window, as well as the ratio of its distance compared to the threshold. Afterwards, ATCQuery uses these window results to construct a result time series. Finally, the result time series is returned to FPB, which then generates CSV files for both the original, and the result series.
5. Evaluation

In this chapter, the ATC algorithm, introduced in the previous chapter, is evaluated in regards to its ability to detect memory leaks. Section 5.1 explains how the algorithm was evaluated and what questions the evaluation should answer. Consequently, Section 5.2 discusses the results, which show that ATC cannot fulfill the set requirements. Therefore, Section 5.3 introduces and evaluates alternative versions of the algorithm. As a result, the modified algorithm can indeed recognize memory leaks, although it does not factor typical patterns of memory leaks in its decision. Finally, Section 5.4 provides a short recap of these findings.

5.1. Evaluation strategy

For the purpose of this evaluation, multiple time series from different systems were available. All of those series consisted of data points, representing the memory usage of the system, with a one minute interval between measurements. Four time series were measured in healthy applications under normal load over a time period of eleven days. Furthermore, seven time series that were ill, i.e. they contained memory leaks, were at hand to evaluate the effectiveness of the ATC algorithm, introduced in Section 4.3. The general aim of this evaluation was to answer the following two questions regarding ATC:

- Can the algorithm distinguish between a time series that contains a memory leak, and one that does not?
- What parameters work best for this purpose?

It turned out early that the reduction technique, used in ATC, was not effective, since the mean distance from a cluster to other clusters was always higher than the standard deviation within a cluster. Consequently, the number of clusters was not reduced at all. Therefore the reduction step was removed from the algorithm to save computing time and was not evaluated further.

The three relevant parameters are threshold, window size, and alphabet size. Since DTW adds the cost of each point in the warp path, as described in Section 4.2.1, it is reasonable to set the threshold in relation to the window size. In the first place, on this account, the threshold was set equal to the window size, in order to be adjusted later. This way, the other parameters could be analyzed more easily.
5. Evaluation

As stated previously, in Section 4.3, the result of ATC is a time series of the ratios between the distance to the best fitting pattern and the threshold for each window in the query series. In order for ATC to distinguish between healthy and ill series, the ratios of healthy series needs to differ noticeably from the ratios of ill series. Particularly, the healthy ratios need to be lower than the ill ratios. However, if this is not the case, it is impossible to choose an accurate threshold that separates healthy and ill queries. Hence, the algorithm was executed, with different alphabet and window sizes, on each of the healthy and ill time series. The results were visualized with the Python script from Section 3.6 and saved with the corresponding CSV files generated by FPB. Furthermore, the mean ratios of each result were averaged again to retrieve the ratios $r_{\text{healthy}}$ and $r_{\text{ill}}$ for each parameter configuration. Additionally, the percentual difference

$$\Delta r = \frac{r_{\text{ill}} - r_{\text{healthy}}}{r_{\text{healthy}}}$$

was determined for each configuration.

However, this metric alone cannot determine the best selection of parameters. Since ATC should provide a visual aid for a human expert, there should be noticeable periods of good or bad results, i.e. the ratio should not switch between extremes in short distances. Therefore, $\Delta r$ was used as a mere aid to select the most promising configurations by hand.

For this evaluation windows of sizes 3, 5, 10, 30, 50, 100, and 200 with alphabets of sizes 3 and 6 were computed.
5.2. Results

Figure 5.1 shows the values of $\Delta r$ across the different parameter configurations. What immediately stands out is that none of the configurations feature a positive value of $\Delta r$, i.e. the ill time series, on average, have a lower mean ratio than the healthy ones. Therefore, it is impossible to choose a threshold value that reliably separates ill from healthy series such that the ratio of healthy series is generally lower than the threshold, while the ratio of ill series is generally higher. Furthermore, the parameters show no recognizable pattern like for example “higher alphabet/window size yields overall a better value for $\Delta r$.”

The only configuration that could produce lower mean values for at least some time series, when comparing healthy to ill, was alphabet size 6 and window size 10. This configuration featured two healthy time series that met this criterion and had a lower mean value than all ill series, i.e. the examples “train58” and “train57”. Figure 5.2 compares the ratios of “train58” (below) and “ill44” (above). As a result of the low window size, in relation to the series length, the algorithm fails to recognize seasonal effects and switches often between extreme ratios.

For instance, although “train58” shows a rising memory usage every afternoon, ATC calculates a higher ratio for every occurrence of this pattern, despite its regularity. Nevertheless, choosing a higher window size does not solve the problem. For example the window sizes 100 and 200 showed lower mean ratios for ill examples across the board.
Further, they also failed to recognize these patterns.

Additionally, higher window and alphabet sizes dramatically increase the computation time of ATC. While these results were computed on a regular home PC (specifications in A.1) in seconds, for low parameters, and minutes, for high parameters, higher window sizes, e.g. 500, take an unfeasible computation time of several hours.

5.3. Alternative approaches

In order to improve the quality of results, produced by ATC, multiple modified versions were evaluated.

5.3.1. Cluster medoids instead of centers

This approach modifies the training step of the algorithm in order to construct a possibly better fitting dictionary for the task. Instead of comparing and returning the cluster centers, i.e. the DBA averages of the elements, this approach uses the medoids, i.e. the elements that are nearest to the respective cluster center. In the literature, this approach known as $k$-medoids [66].

Figure 5.3 shows the results for the k-medoids approach. Except for one outlier at
5.3. Alternative approaches

Figure 5.3.: $\Delta r$ for the medoids approach with different parameters

window size 10, this method shows better overall results with alphabet size 3. Additionally, it yields a positive value of $\Delta r$ for higher window sizes, e.g. up to 10% at window size 200. For this window size, three of the healthy examples feature a lower mean ratio than all of the ill examples, while one healthy example ties with the ill example that has the lowest mean ratio.

5.3.2. Sliding window instead of motifs

As the previous approaches showed a relatively low difference between the ratios of ill and healthy time series, the idea occurred that the training step may produce too generic patterns, i.e. patterns that fit on a majority of memory usage time series regardless of the presence of memory leaks. Therefore, this thesis tried an additional, more simplistic approach. Instead of clustering motifs, the resulting subsequences of a sliding window approach were clustered.

Since this version of ATC did not use motifs, it could no longer initialize the clustering algorithm with the number of found motifs. As a result, the alphabet size was used to determine the number of clusters and the clusters were initialized randomly.

The number of clusters, i.e. “#clusters” in Figure 5.4, was evaluated for the values 2, 3, 5, 10, and 20.

Figure 5.4 shows the value of $\Delta r$ across different parameters. As in the medoids approach, the window size 200 seems to produce a good value of $\Delta r$ (the gray bar with
5. Evaluation

However, when examining the results by hand, it turned out that the quality varied drastically across examples. While “train57” had a lower mean ratio than any of the ill examples, “train56” had a higher mean ratio than all of them, but two. Additionally, Figure 5.4 shows that higher window sizes generally seem to produce a better result. In contrast, the distribution of $\Delta r$ across different numbers of clusters shows no noticeable pattern. In conclusion, this approach only yields a highly unstable result.

### 5.3.3. Combining the approaches

Additionally, the previous modifications were combined in a third approach that clusters sliding window subsequences with a medoids based algorithm.

Figure 5.5 shows results which suggest that window sizes lower than 30 are unsuitable. Furthermore, the effect of varying numbers of clusters on the quality of the result seems to follow no recognizable pattern. However, since the clustering was initialized random in this approach, the randomness could also have skewed the data. Based on this assumption, this thesis suspects that 20 and 5 clusters yield the best results.

The result with the highest value of $\Delta r$, i.e. with 20 clusters and window size 100, improves the previous results considerably. All healthy examples have mean ratios that are between 10% and 48% lower than these of every ill example.
A closer consideration of two examples from this configuration gives further insight. Figure 5.6 shows the ratios of examples “train58” and “ill44” again, calculated with the combined approach. First of all, analysis of all ratios showed that healthy examples generally feature a larger range of ratios than the ill ones. For instance, “train58” has ratios between 0 and 0.56, while “ill44” shows ratios only between 0.32 and 0.52.

Another observation is that the ratio of “ill44” does not increase noticeably, although the memory consumption rises continually. Furthermore, the algorithm still fails to notice the seasonal patterns in “train58”. Finally, the segments with decreasing memory usage of “ill44”, e.g. between 14:00 and 16:00, lead to a higher ratio, which is unintuitive.

Despite these shortcomings, a threshold exists that separates healthy from ill examples with this method. With consideration of the generated data, this threshold was set to $t = 0.35 \cdot \text{windowsize}$.

Figure 5.7 shows both series colored, depending on their ratio compared to the threshold. However, this figure shows the values of the time series, instead of the ratios calculated by ATC. Like in Section 4.3, green indicates normal, yellow indicates distances that are over but within 10% of the threshold, and red indicates anomalies.
5.4. Overall results

The vanilla ATC algorithm turned out to produce poor results. However, the combination of multiple adjustments to the algorithm presents an improvement. In detail, these adjustments where the use of medoids as patterns, instead of DBA averages, and a sliding window approach, instead of motif search, to retrieve these patterns.

Window sizes of 100 and 200, alphabet sizes, i.e. number of clusters for the modified approach, of 20 and 5, and threshold $t = 0.35 \cdot \text{windowsize}$ have been shown to yield the best results. Although ATC does not distinguish between ill and healthy time series like humans do, i.e. ignores residual patterns and increasing overall usage, it can still noticeably differentiate between them.
5.4. Overall results

Figure 5.7.: Colored examples for the result presented in Figure 5.6
6. Conclusion

In order to assess dynamic behavior of software under development, in a CI environment, run-time metrics need to be measured and analyzed. Further, visual feedback is necessary to allow a quick interpretation of the results.

Chapter 2 introduced CI and the related practices Continuous Deployment and Continuous Delivery. The tools, introduced in Section 2.2, provide the necessary means to integrate dynamic measurements and analyzes in a CI environment.

This is accomplished with a toolchain, which was described in Chapter 3. It contains the CI environment in a VM that is automatically set up (Section 3.2). The CI server inside the VM provides a reproducible environment with an exchangeable configuration (Section 3.3). In this environment, the regular build process is extended with three additional steps. First, Section 3.4 introduced the gatling-docker plugin, which executes stress tests using Docker containers. Second, the dfd-solr-jenkins then stores the results of the former plugin in a database. Finally, a third step executes machine learning analyzes on the run-time metrics in the database and provides visual results inside the environment.

In Chapter 4, Machine Learning (Section 4.1) and algorithms for time series analysis (Section 4.2) were explained. Furthermore, Section 4.3 introduced the new anomaly detection algorithm ATC that tries to detect anomalies, by detecting normal behavior to rule out anomalies. ATC takes a time series and extracts frequently occurring patterns, i.e. motifs of a specific length from it. Then, it clusters these motifs with a k-means algorithm and takes the average time series of each cluster as a dictionary for normal behavior. Finally, it applies this dictionary on new time series, resulting in another time series that contains scores for each point in the original series.

Finally, Chapter 5 evaluated the algorithm, with different parameters, for its ability to distinguish time series with memory leaks from such, without. As a result, ATC was not able to accomplish this, though an alternative version of ATC had more success. This version used a sliding window approach to receive the patterns and additionally created the dictionary from the medoids, i.e. the element nearest to the average, in the clusters. Although the modified version recognized leaks and non-leaks different, it failed to notice characteristic patterns in the time series.
6. Conclusion

6.1. Future work

The ATC algorithm, as well as the developed plugins, are rather prototypical and can be improved in various ways. Therefore, this section discusses possible next steps for the CI toolchain and the ATC algorithm.

6.1.1. Improvements for the CI toolchain

In order to analyze the behavior of the currently built version of the developed software, JMX [37], or similar tools, could be integrated into the toolchain. This could be accomplished by developing a third Jenkins plugin that measures JMX data and stores the results in a Solr instance.

The gatling-docker plugin represents rather a proof of concept than a finished software. In particular, it built the foundation that allows a flexible stress test engine powered by Docker containers. Although the plugin already supports the parallel execution of simulations as well as scaling with Docker, this feature is very resource intensive on a single machine. However, the plugin could be extended to distribute Docker containers across multiple hosts.

As the plugin already communicates with Docker daemon via TCP, it could easily be extended to control a Docker daemon on a different machine, or multiple daemons on multiple machines as well. The data transfer could be managed with a private Docker repository, where the particular machines could upload their data containers as result images, which in turn are downloaded from the main host to collect the results. However, security would become a factor, as it is no longer possible to allow only connections to the Docker daemon from localhost.

Another approach could implement support for Mesos [3] clusters. As Mesos supports Docker, the plugin could outsource the scaling process to Mesos and just provide a single container. However, this approach has to solve the problem of the data transfer, too. Since a Mesos cluster cannot just link host folders into the containers, because it is not necessarily determined which host executes the container. Nevertheless, if these problems can be solved, gatling-docker can provide a powerful stress test engine for large scale performance testing.

Moreover, the utilized Docker images could be adjusted to simulate a unique IP address, which is a feature that is not possible with Gatling alone.

6.1.2. Future work for ATC

Chapter 5 showed that ATC could not sufficiently distinguish between healthy and ill time series. Although the modified approach accomplished this, it still failed to recognize seasonal patterns and a steady increase in memory usage. Therefore, this thesis suggests
6.2. Related work

Analysis of software metrics with machine learning methods has been attempted rarely. Nevertheless, a few examples are available in the literature.

The PhD thesis [8] introduced the performance monitoring and analysis plugin $\Theta$PAD for Kieker [46] that performed online anomaly detection on dynamic metrics of a social network platform, i.e. it continually received measurements over time, constructed time series, and analyzed them. Later, the Master’s thesis [23] developed another plugin, called $\Theta$PADx, which extended the original approach by an improved detection of long term anomalies.

However, these tools were not developed as stand-alone applications and practice analysis of a production system, in contrast to this thesis that focuses on analysis of software under development.

Furthermore, [49] specifically tried to detect memory leaks with machine learning methods, but instead of analyzing run-time metrics, they extracted heap objects and analyzed them for staleness similarity. The general assumption behind this approach was that a leaking object differs in staleness, compared to non-leaking objects, noticeably. To achieve this, Support Vector Machines were utilized and yielded promising results at the cost of a rather long training time, i.e. several days.

Additionally, [64] used statistical and classification methods to detect memory leaks in Java applications also based on object staleness. They could improve accuracy over regular statistic approaches by up to 7%.

Finally, [72] introduced FPB, a stand-alone application for machine learning analysis that was already mentioned in Section 4.3. In addition, [72] applied Hidden Markov Models and ARIMA in order to detect memory leaks. Although those methods could detect memory leaks, they yielded a high number of false positives.

General time series analysis has many applications including medical analysis, where ECG data is analyzed [10]. Anomaly detection is commonly attempted with classification, clustering, or prediction methods. For instance, [55] applied the ARIMA model successfully to find anomalies in network traffic data.

This thesis already introduced DTW in Section 4.2, which was used for classification,
6. Conclusion

e.g. [83], and clustering, e.g. [70]. Additionally, there exist various improvements to DTW like the discussed FastDTW [73]. [69] further introduced drastic optimizations to the algorithm that could speed up DTW to the run-time of ED for large datasets. Although a k-means algorithm that uses both DTW and time series motifs as it is introduced in Section 4.3 was not found in the literature. Nevertheless, k-means with DTW [67] and motifs [44] separately has achieved positive results, although both publications had another scientific focus.

The basic idea behind ATC was based on [16], where an algorithm was introduced that uses patterns in a dictionary-like way to detect anomalies. Furthermore, [12] detected anomalies (or novelties, how they called it), based on ideas from Immunology, also with a threshold based method.

Methods to retrieve time series motifs are also frequently investigated, e.g. online discovery [58], parallel discovery [62], or disk-aware discovery [59]. Beside the fixed length approach of SAXMotifs [51] [74], introduced in Section 4.2.2, exists the concept of variable-length motifs. Various publications proposed methods to detect these motifs, e.g. [50] introduced a grammar based approach using SAX. In order to find variable-length motifs faster, [57] optimized the brute force approach, i.e. calling a fixed-length motif algorithm for every possible length, and achieved a speed up of 23. Similar to motifs, [60] use unsupervised shapelets to cluster time series. These shapelets are created by extracting only local patterns and ignoring the rest of the data.

Another relevant concept is that of discords. While motifs are frequently occurring, similar patterns in time series, discords are subsequences that are highly dissimilar to many or all other subsequences in the series. In the publication [45], discords are introduced and the SAX trie structure, described in Section 4.2.2, is used to detect them. Additionally, [84] introduces an efficient disk-aware algorithm for discord detection.

There is a Gatling plugin [24] for Maven that can be used in a CI environment, although it does not support parallel execution of simulations and cannot rely on the flexibility of Docker for future improvements, like the approach of this thesis. Although there are no further similar approaches for the integration of dynamic measurements and analyzes in CI, the article [28] is certainly related. It describes how Google manages testing at a very large scale in a CI environment.
Bibliography


Bibliography


[31] *HTML Publisher Plugin*. Available at: https://wiki.jenkins-ci.org/display/JENKINS/HTML+Publisher+Plugin, retrieved on Oct. 11, 2014.


A. Appendix

A.1. Specifications of the evaluation system

- Processor: 4x Intel(R) Core(TM) i5-2400 CPU @ 3.10GHz
- Memory: 8159MB
- Operating System: Linux Mint 17 Qiana 64-bit
- Linux Kernel: 3.13.0-24-generic
- Graphics Card: Radeon HD 7770/8760/R7 250X