BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INTELLIGENT SYSTEMS
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

NEW MODELS FOR AUTOMATIC DETECTION OF PERFORMANCE DEGRADATION

NOVÉ MODELY PRO AUTOMATICKOU DETEKCI DEGRADACE VÝKONU

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR **AUTOR PRÁCE**	**ŠIMON STUPINSKÝ**
SUPERVISOR **VEDOUCÍ PRÁCE**	**Doc. Mgr. ADAM ROGALEWICZ, Ph.D.**

BRNO 2019

Ústav inteligentních systémů (UITS) Akademický rok 2018/2019

Zadání bakalářské práce

22118

Student:	**Stupinský Šimon**
Program:	Informační technologie
Název:	**Nové modely pro automatickou detekci degradace výkonu**
	New Models for Automatic Detection of Performance Degradation
Kategorie:	Algoritmy a datové struktury

Zadání:

1. Seznamte se s projektem Perun (správcem výkonnostních profil) a s metodami profilování programů.
2. Prostudujte metody modelování výkonu programů založeného na statistických metodách (regresní analýza, neparametrické jádrové odhady, apod.).
3. Navrhněte a implementujte model v nástroji Perun pro tvorbu modelů výkonu programů na základě nasbíraných profilovacích dat.
4. Navrhněte a implementujte metodu pro automatickou detekci degradace výkonu založenou na nově implementovaných modelech.
5. Demonstrujte řešení na netriviální sadě příkladů.

Literatura:

- Pavela, Jiří. *Knihovna pro profilování datových struktur programů C/C++*. Brno, 2017. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií.
- Smoothing and Regression: Approaches, Computation, and Application
- Oficiální stránky projektu Perun: https://github.com/tfiedor/perun

Pro udělení zápočtu za první semestr je požadováno:

- podle domluvy s vedoucím

Podrobné závazné pokyny pro vypracování práce viz http://www.fit.vutbr.cz/info/szz/

Vedoucí práce:	**Rogalewicz Adam, doc. Mgr., Ph.D.**
Vedoucí ústavu:	Hanáček Petr, doc. Dr. Ing.
Datum zadání:	1. listopadu 2018
Datum odevzdání:	15. května 2019
Datum schválení:	1. listopadu 2018

Abstract

Performance testing is a critical factor in the optimisation of programs during its development, but it is still not so well developed in comparison to functional testing. A framework PERUN provides full automation of performance management, thereby contributing to the development of this area. We have introduced three non-parametric approaches to performance data modelling: regressogram, moving average and kernel regression, which were integrated within this framework. We try to achieve appropriate approximations of performance data using these techniques, without the assumption of dependence between two variables, which represents the main advantage in comparison to parametric techniques. Further, we have proposed and implemented two methods for automatic detection of performance changes, which works with all kinds of models within PERUN. We have demonstrated our solutions on the real project (VIM), and on the set of the experimental cases, in which we compared proposed solutions with existing. We have achieved decreased time processing about two-thirds and an almost triple improvement in the fitness of data modelling with new modelling approaches. The proposed detection methods detected performance degradation of three specific functions in comparison of two different versions of VIM, where was present a known performance issue.

Abstrakt

Testovanie výkonu predstavuje kľúčový faktor pri optimalizácii programov počas ich vývoja, avšak, v súčastnosti nie je vyvinutý na takej úrovni ako funkcionálne testovanie. Nástroj PERUN poskytuje automatickú správu výkonnosti programov, čím prispieva k vývoju tejto oblasti. V tejto práci predstavujeme tri neparametrické prístupy modelovania výkonnostných dát: regresogram, kĺzavý priemer a jadrové odhady, ktoré boli integrované v rámci tohto nástroja. Použitím týchto techník sa snažíme dosiahnuť vhodnú aproximáciu výkonnostných dát bez predpokladu závislosti medzi dvoma premennými, čo predstavuje hlavnú výhodu oproti aktuálne používaným parametrickým technikám. V rámci tohto nástroja, sme tiež navrhli a implementovali dve metódy pre automatickú detekciu degradácie výkonu, ktoré dokážu pracovať so všetkými druhmi modelov. Riešenie sme demonštrovali na reálnom projekte (VIM) a na sade experimentálnych prípadov, v ktorých sme navrhnuté riešenia porovnali s už existujúcimi. Novými prístupmi modelovania sme dosiahli zvýšenú časovú efektivitu o dve tretiny a priemerne trojnásobne lepšiu presnosť modelovania dát. Navrhnuté metódy detekovali degradáciu výkonu troch špecifických funkcií v porovnaní dvoch verzií programu VIM, kde bola prítomná ohlásená výkonnostná chyba.

Keywords

performance changes, automatic detection, regressogram, moving average, kernel regression

Kľúčové slová

výkonnostné zmeny, automatická detekcia, regresogram, kĺzavý priemer, jadrové odhady

Reference

STUPINSKÝ, Šimon. *New Models for Automatic Detection of Performance Degradation.* Brno, 2019. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Doc. Mgr. Adam Rogalewicz, Ph.D.

Rozšírený abstrakt

Výkonnostné testovanie je rozhodujúcim faktorom pri optimalizácií programov počas ich vývoja. Cieľom tejto formy testovania je stanoviť výkonnosť systému za určitých podmienok a identifikovať jeho kritické miesta. Testovanie výkonnosti ako súčasť vývojového cyklu, napríklad v rámci priebežnej integrácie, však stále nie je tak dobre vyvinuté v porovnaní s funkcionálnym testovaním. Udržiavanie optimálneho výkonu počas vývoja si vyžaduje sledovanie viacerých, často protichodných aspektov za rýchlo meniacich sa podmienok.

Hoci existuje niekoľko kvalitných nástrojov na testovanie výkonnosti, plnú automatizáciu poskytujú len niektoré. Pri správe výkonnostných profilov bez automatizácie je užívateľ nútený manuálne spravovať údaje o výkone svojho programu. Manuálna manipulácia s veľkým množstvom údajov je však náchylná na chyby a môže viesť k strate histórie o výkonnostných zmenách. Preto by bolo vhodné nájsť systém, ktorý bude distribuovaný a schopný spravovať výkonnostné profily s najväčšou možnou automatizáciou.

Pre správu výkonnosti rôznych verzií programov vyvinula výskumná skupina VeriFIT nástroj Perun. Tento nástroj spravuje výkonnostné profily ktoré zodpovedajú rôznym verziám programov a poskytuje sadu modulov vhodných na automatizáciu výkonnostných regresných testov, následné spracovanie existujúcich profilov a interpretáciu výsledkov (napr. detekciu zmien alebo vizualizáciu). V súčasnosti tento nástroj zahŕňa rôzne druhy kolektorov, post-procesory poskytujúce výkonnostné modely získané regresnou analýzou a niekoľko metód pre automatickú detekciu výkonnostných zmien.

Spracovanie profilov, ktoré obsahujú nazberané výkonnostné dáta, je jedným z najcitlivejších aspektov, ak chceme dosiahnuť presnú analýzu výkonu. Post-procesory vytvárajú rôzne druhy predikčných modelov, ktoré následne slúžia ako vstupy pre metódy detekcie či ďalšiu interpretáciu používateľom. V súčasnosti dostupný post-procesor implementuje parametrický prístup nazývaný regresná analýza, ktorý však vyžaduje poznať takzvanú nezávislú premennú. Tento prístup odhaduje hodnoty závislej premennej (napr. doba behu funkcií) pre každú hodnotu nezávislej premennej (napr. veľkosť dátovej štruktúry), teda skúma vzťah medzi týmito dvoma premennými. Tento predpoklad, že neznáma, analyzovaná funkcia patrí do triedy funkcií závislých na parametroch, nie je však vždy splnený: nezávislá premenná môže jednoducho chýbať alebo môže byť neznáma.

Hlavným prínosom tejto práce je rozšírenie nástroja Perun o nové komponenty, ktorých úlohou je dosiahnutie presnejších výsledkov v automatickom procese detekcie výkonnostných zmien. V tejto práci uvádzame tri nové post-procesory, ktoré implementujú tri rôzne neparametrické techniky modelovania: regresogram, kĺzavý priemer a jadrové odhady. Tieto techniky prinášajú nové možnosti modelovania výkonnostných dát a dopĺňajú uvedené nevýhody parametrických prístupov. Taktiež sme navrhli dve nové metódy detekcie, ktoré boli vytvorené pre všetky druhy modelov, ktoré sú vytvorené postprocesormi v rámci nástroja Perun, t.j. pre parametrické aj neparametické modely.

Riešenie sme demonštrovali na reálnom projekte (Vim) a na sade experimentálnych prípadov, v ktorých sme navrhnuté riešenia porovnali s už existujúcimi. Nové prístupy modelovania dosiahli zvýšenú časovú efektivitu o dve tretiny a priemerne trojnásobne lepšiu presnosť modelovania dát. Navrhnuté metódy detekovali degradáciu výkonu troch špecifických funkcií v porovnaní dvoch verzií programu Vim, kde bola prítomná ohlásená výkonnostná chyba. Výsledky, ktoré sme dosiahli sú veľmi povzbudivé a ukazujú vysoký potenciál nášho prístupu pri automatickej detekcii výkonnostných zmien. Výsledky tejto práce boli tiež prezentované na študentskej konferencii EXCEL@FIT'19, kde sme boli ocenení jedným z generálnych sponzorov tejto konferencie, za výnimočnú prácu s veľkým prínosom do praxe.

New Models for Automatic Detection of Performance Degradation

Declaration

Hereby I declare that this bachelor's thesis was prepared as an original author's work under the supervision of Doc. Mgr. Adam Rogalewicz, Ph.D. The supplementary information was provided by Ing. Tomáš Fiedor and Mgr. Bc. Hana Pluháčková. All the relevant information sources, which were used during preparation of this thesis, are properly cited and included in the list of references.

....
Šimon Stupinský
May 15, 2019

Acknowledgements

I thank my supervisors from VeriFIT performance team — Tomáš Fiedor, Hana Pluháčková, Adam Rogalewicz, Tomáš Vojnar. I thank for the support received from Red Hat company and H2020 ECSEL project Aquas.

Contents

Chapter 1

Introduction

"Just as athletes can't win without a sophisticated mixture of strategy, form, attitude, tactics, and speed, performance engineering requires a good collection of metrics and tools to deliver the desired business results."

– Todd DeCapua

Performance testing (and profiling) is a critical factor for the optimisation of programs during its development. This form of non-functional testing aims to determine the performance of a system under certain conditions and to identify its critical locations. However, performance testing as a part of the development cycle, e.g. integrated in continuous integration, is still not so well developed compared with functional testing. Maintaining optimal performance during the development requires tracking multiple, often conflicting, aspects under rapidly changing conditions.

Although there exist several high-quality tools for performance testing, the full automation of the profiling resources or the subsequent comprehensive management of the created performance is provided only by few of them. When managing performance profiles without any automation, the user is forced to annotate and manage all performance data manually. However, manual manipulation with a large amount of data is highly prone to error and may lead to a loss of the exact history of tracked changes.

Hence, developers would like a system, that would be light-weight, distributed and able to manage collected data and performance profiles as automatically as possible. To manage the performance of program versions the research group VeriFIT developed the framework PERUN: *Performance under control* [11]. It manages performance profiles corresponding to different versions of projects and offers a tool suite suitable for automation of the performance regression tests, post-processing of existing profiles and practical interpretation of the results (such as detection of changes or visualisation). Nowadays, this framework involves different kinds of collectors, post-processors providing performance models obtained by the regression analysis and several detection methods.

A post-processing of profiles is one of the most delicate aspects if one wants to achieve the most accurate performance analysis. The post-processors create different kinds of prediction models that can serve as inputs for change detection methods or for further user interpretation. The currently available post-processor implements a parametric approach called regression analysis that, however, requires to have a so called independent variable. This approach predicts values of the dependent variable (e.g. run-time of functions) for every value of the independent variable (e.g. the size of the underlying structure), i.e. it explores the relationship between them.

However, the assumption of the regression-analysis, that an unknown, analysed function belongs to the class of functions dependent on some parameters, is sometimes not fulfilled: the independent variable can simply be missing, completely unknown or non-existent at all. In that case, a constructed model will not be accurate and therefore it will affect the subsequent analyse performed, e.g., by detection methods. We need to have and an excellent model to achieve adequate results under any conditions. Hence, we propose to shift to non-parametric modelling methods, which can potentially achieve more accurate results and do not need any independent variable.

In this thesis, we present three new post-processors, which implement three different non-parametric modelling techniques: regressogram, moving-average and kernel-regression. These techniques bring new possibilities of performance modelling and complement the mentioned disadvantages of the parametric approaches. Further we propose two new change detection methods, namely: Integral-Comparison and Local-Statistics. Our proposed methods were designed in general for all kind of models, that are created by individually post-processors in framework Perun, i.e. both for non-parametric and parametric.

The proposed solutions were tested on Vim repository, with an aim to show that they can discover known performance bugs. We tested new non-parametric models in combination with implemented detection methods, as well as compared with existing methods within framework Perun. We achieved quite encouraging results, which have shown high potential of our approach in automatic detection of performance changes. The results of this thesis were also presented at students conference Excel@FIT'19, where we got the award from the general sponsor of this conference for extraordinary work with great benefits to the practice.

Document Structure. First and foremost, we briefly discuss the architecture of framework Perun in Chapter 2, which also introduces the reader into the automatic performance management process. Further, Chapter 3 gives some preliminaries in performance data modelling and the involvement of non-parametric methods in this process. In Chapter 4, we introduce a design and implementation details of proposed post-processors, which are based on the selected non-parametric methods. Following this, in Chapter 5, we give a detailed account to proposed detection methods including their algorithmic description. Furthermore, Chapter 6 provides an experimental evaluation of the proposed detection methods, primarily using new non-parametric models, and a comparison of these methods with existing methods in this framework. Finally, a summary of the acquired result and future ideas of our framework can be found in Chapter 7.

Chapter 2

Perun

PERUN (Performance Under Control) [11] is an open source light-weight Performance Versioning System, which was invented by VeriFIT research group. PERUN tries to achieve the full automation process of performance management and thereby takes care of managing performance profiles (i.e. the set of collected performance records) for a broad range of project types. It provides the full functionality for automation of the profiling process, maintains and stored collected data, and allows efficient processing of the results, as well as a set of visualisation techniques. PERUN serves as a wrapper over the existing version control systems, and is primarily inspired by these particularly distributed version-control systems (e.g. `git` or `SVN`), on whose basis are designed e.g. the CLI[1] or the data storage. The main reason for the similarity is that one of the goals of PERUN is to store the information about performance with focus on history of the project development. Each instance of PERUN is integrated into existing version-control systems, which allows one to automatically maintain and manage the performance profiles for different versions of a project, without the need for manual user involvement.

This thesis extends the PERUN framework by few individual components. The resulting components, presented in this work, were integrated into PERUN (the integration of some components, however, is not yet fully merged into PERUN's master branch). The following sections will briefly introduce the architecture and the interface of PERUN. This Chapter is based on [17, 9, 18] and PERUN documentation [11].

2.1 Architecture

As shown in Figure 2.1, the internal architecture of PERUN can be logically divided into the following components — data, logic, viewers and checkers.

Data. This unit contains the modules that defines format of performance profiles, and wrappers over the existing version control systems supported in this framework (currently only git). Each of the listed components works with a profile, and therefore profile is a primary data unit in this framework.

Logic. Logic unit includes creating and management of individual profiles and contains two principal types of components. First of them, called collectors, performs the collection of the data and subsequently creates of profiles from these collected data. The second one

[1]Command Line Interface

processes profiles for further interpretation (visualisation and detection) and therefore are called post-processors. The Chapter 4 proposes these new post-processors implementing non-parametric methods that are part of this work and have been integrated into PERUN.

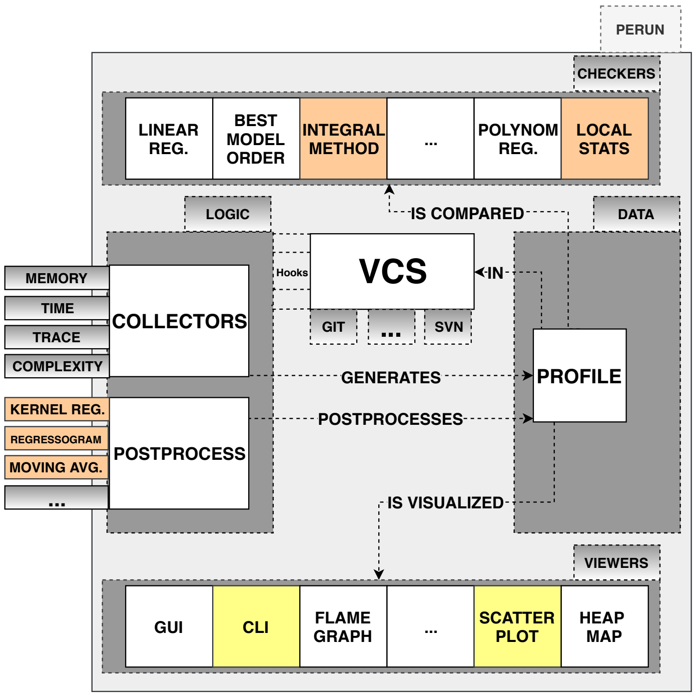


Figure 2.1: PERUN architecture; its logical, data, visualisation and checkers components, including their respective modules. The arrows indicate the mutual communication between the individual modules. The scope and results of this thesis are marker coloured — new modules in orange and modified modules in yellow.

Viewers. It is partly independent component which primary objective is to provide the different interpretations of data to a user, especially the visual one. A user can choose from the supported various visualisation techniques (see Figure 2.1), which gives him a better interpretation of the data, compared to raw data. Next, it contains a wrapper for the graphical and command-line interface. In Chapter 4, we will introduce how to visualise non-parametric models that are relevant in the context of this work.

Checkers. The last component includes detection methods, which compare the profiles pair and reports possible changes in performance. A checker component provides different kinds of detection methods, which depend on the individual section of the profiles (raw data, parametric models, non-parametric models). The input of these methods are two profiles and its output is the report about detected changes. Chapter 5 describes novel detection methods primarily for non-parametric models.

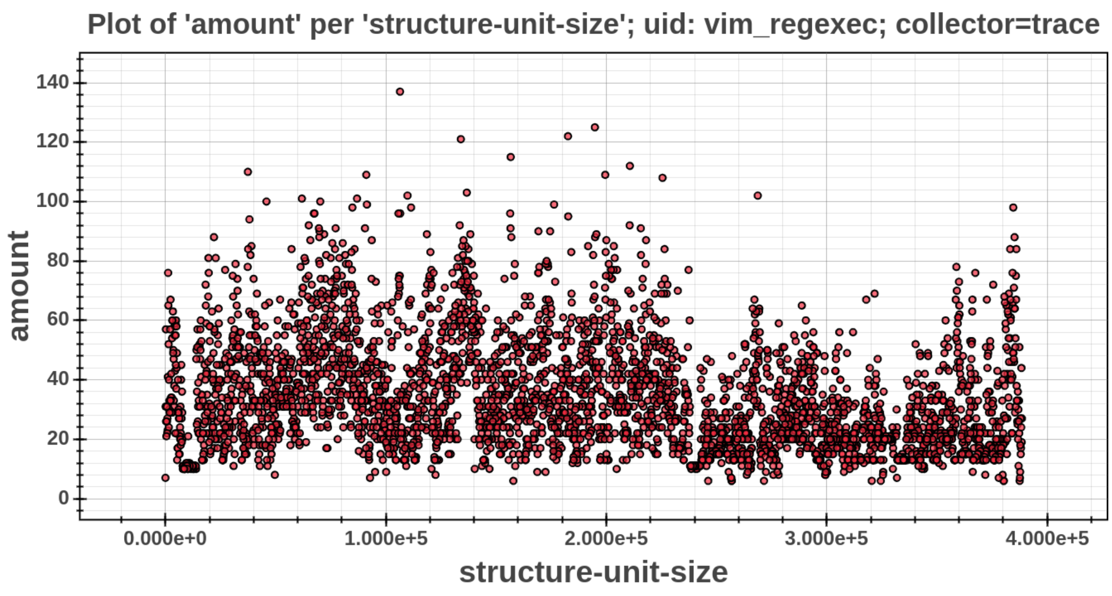


Figure 2.2: The profiling data collected by trace collector on VIM binary, in particular the function `vim_regexec` and its measured values at individual calls. The `amount` values represents the time consumption of this function during the program execution.

2.2 Collecting Performance Data

The collectors (see Figure 2.1) collects the performance data usually from the binary or other executable of the current project. Most of the collectors collect the data based on the lightweight instrumentation of the profiled program. PERUN currently offers four collectors (trace, time, memory and complexity), that are independent and focus on tracing of the different aspects in the programs [8]:

1. **Trace Collector** focuses on run-times of functions (C or C++ language) that are executed during the collection. It realise the profiling data collection by injecting probes at specified code locations (such as functions entries and exits). The Figure 2.2 shows the profiling data collected by this collector.

2. **Memory Collector** collects the allocation specification in C or C++ functions, type of allocations, or target addresses of allocations. An example of interpretation may be the consumption of allocated memory (Y-variable) depending on the allocation order (X-variable) at manipulation with this structure.

3. **Time Collector** is a wrapper over the `time` utility. It collects overall running times of arbitrary commands from the profiled program.

4. **Complexity Collector** collects running times C/C++ functions along with the size of the structures they were executed on. For example, it can capture the dependency of time consumption (Y-variable) depending on estimate of the size of the structure (X-variable).

2.3 Post-Process Performance Data

The post-processors process the performance data obtained in the collection phase. As shown in the Figure 2.2, raw performance data do not have usually significant value without further processing. Post-processors e.g. create different kinds of models by the transformation of these data and thereby provides data suitable for difference analysis, degradation detection and manual interpretation. PERUN currently supports the following selected post-processors:

1. The **Clusterizer** implements a simple heuristics that classifies the collected data into clusters according to the similarity of its values. It provides two simple strategies to cluster the data: by sort order of values or by sliding window. Both strategies allow one to derive and estimate the so-called independent variable that is needed to find the regression data model. It is therefore especially useful in conjunction with regression analysis post-processor.

2. **Regression Analysis** post-processor implements the statistical parametric techniques to determine the relationship between a dependent (Y) and independent (X) variables. The result of this analysis is the set of the mathematical functions $y = \hat{\beta}_1 f(x) + \hat{\beta}_0$ that describe the behaviour of code functions in the analysed program (e.g. the dependency of consumed memory to the size of a data structure).

$$\hat{\beta}_0 = \frac{\sum y_i - \hat{\beta}_1 \sum f(x_i)}{n} \tag{2.1}$$

$$\hat{\beta}_1 = \frac{n \sum f(x_i) y_i - \sum f(x_i) \sum y_i}{n \sum (f(x_i))^2 - (\sum f(x_i))^2} \tag{2.2}$$

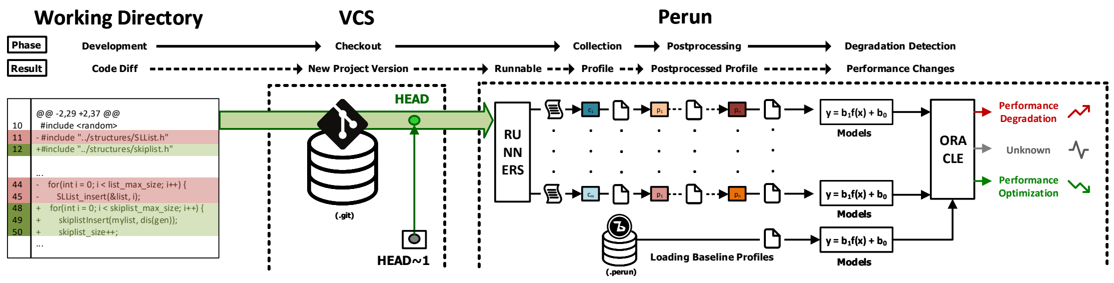
   This post-processor currently supports the selected types of regression models: constant, linear, logarithmic, exponential, power and quadratic. All models, except quadratic, have two coefficients and therefore they use the general Formula 2.1. Formula 2.2 represents the computation of coefficient for the quadratic model, that requires more coefficients than remaining models. The regression analysis will be described in more details in Section 3.1.

Post-processors can be used in conjunction to obtain appropriate models for method for detecting performance changes from any collected data. We assume that the dependent variable (Y) represents the run-time of a function for the following example. First, we use a clusterizer to estimate the values of independent variable for each measured resource resulting into relevant classified clusters. Subsequently, a regression-analysis post-processor estimates these resources with the regression performance model, which predict the values of run-time for a different amount of clusters. Note that for such estimation, we need some independent variable (X), such as the size of the input data or actual values of function parameters, hence we use the clusterizer to get some estimate.

It is due to the reason that a regression-analysis post-processor performs the analysis based on parametric techniques. These techniques assume the normal distribution of data, so its density will have a bell-shaped curve, and the parameters such as mean and variance are known. However, the distribution of the data is not known always in our case of performance modelling, and therefore the non-parametric estimates may be appropriate. These estimates do not need to know the parameters and coefficients that characterise the data to its properly estimate. Chapter 4 discusses the possibilities of post-processing of profiles by non-parametric techniques in more detail.

2.4 Detection of Changes

As shown in Figure 2.3, the last step in one of the chains automation process of performance management is the detection of changes. Current detection methods perform the difference analysis between the models created in post-process phase by individual post-processors and subsequently detect potential changes. The input of detection is a pair of profiles, from which are obtained relevant models for analysis. The first profile (so-called target profile) represents the newly released version of the project. The second profile (so-called baseline profile) represents the stable version of the project against we will compare the target profile. The baseline usually serves as an expectation how the program should perform.



```
@@ -2,29 +2,37 @@
 #include <random>
- #include "../structures/SLList.h"
+#include "../structures/skiplist.h"

...
-    for(int i = 0; i < list_max_size; i++) {
-        SLList_insert(&list, i);
+    for(int i = 0; i < skiplist_max_size; i++) {
+        skiplistInsert(mylist, dis(gen));
+        skiplist_size++;
...
```

Figure 2.3: Overview of the automatic detection method of performance changes over the project repository. This thesis focuses mainly on post-processing methods and detection of changes [11].

All detection methods returns the structure `DegradationInfo` with information about detected changes. This structure contains the following members:

1. `Result` member contains the type of potential performance changes, which can be one of the follows: *Degradation*, *Maybe Degradation*, *No Change*, *Maybe Optimisation* and *Optimisation*. The type depends on the particular detection methods and its thresholds.

2. `Error Rate` denotes how significant a change has occurred in the target profile in comparison to the baseline profile, what represents the value of absolute error (ε_{rel}) or relative error (ε_{abs}).

3. `Confidence` can help to decide whether the changes are worthy of fixing and if the change is real or spurious. This member contains the pair, that includes confidence type and confidence value, where the type can be represented for example as the coefficient of determination R^2.

4. `Severity` of the changes is reported by detection methods, that analyse the regression models created by regression-analysis post-processor. The specific kind of the regression model (quadratic, logarithmic, etc.) that is represented by members `From` and `To` in this structure.

These crucial aspects, provided for each detected change, help achieve a high ratio of performance fixes. However, all detection methods depend on the models created by individual post-processors, and therefore the fitness of the model affects the final result. In the Chapter 5 we will introduce the new detection methods, that can work with all types of models created by individual post-processors.

Chapter 3

Performance Data Modelling

In this chapter, we introduce the area of the performance data modelling and its related theory. First, we discuss the basic concepts of statistical techniques for performance modelling, including the comparison of two main approaches: parametric and non-parametric. Then, we describe three non-parametric modelling methods — regressogram, moving average and kernel regression.

3.1 Smoothing

Smoothing is a statistical technique to modelling real functions based on observed or collected data. The goal is to find the estimate (appropriate approximation) of the unknown function to filter out random fluctuations and to provide a better understanding of the data structure. The estimates can determine any type of functions, for example, the functions that describe the behaviour of the data: density, distribution function and regression function, that we use in the field of the performance modelling.

The raw performance data contains pairs consisting of values from a set of independent variables X (such as workload size) and values from a set of dependent variables Y (such as function time consumption). If we want to interpret these data, we need to find appropriate function describing the relation between these variables. The solution of this task is the fitness of the appropriate curve (regression curve) to collected data-set of points from certain program runs. We assume the standard regression model, which can be formalised as:

$$Y_i = m(x_i) + \epsilon_i, \quad i = 1, \ldots, n, \tag{3.1}$$

where m is unknown regression function, $x_i, i = 1, \ldots, n$, are the data-set points and $\{\epsilon_i\}$ is the set of measurement errors. We assume that these noise[1] (i.e. the measurement error) is independent and identically distributed (iid), centred, with variance σ^2:

$$\epsilon_i \overset{\text{iid}}{\sim} N(0, 1), \quad i = 1, \ldots, n. \tag{3.2}$$

There exist two possible ways how to characterise the task of finding the unknown function: parametrically or non-parametrically. Parametric estimates assume that the unknown function belongs to a class of functions that depend on parameters and its primary objective is to find these parameters. The purely parametric approach does not always

[1]Noise in smoothing area is unexplained variability within a data sample.

meet the needs of flexibility, but still, it is useful and retains its benefits, such as trustworthy results with skewed and non-normal distributions or when the groups have different amounts of variability. The example of a parametric estimate of regression function can be the regression curve reflecting the linear dependency of a run-time depending on the size of the underlying structure. On the contrary, non-parametric estimates do not assume that an unknown function has the prescribed shape, only assume smoothness of the estimated function (i.e. sufficient number of continuous derivatives). The non-parametric approach is related to increasing data processing requirements, whether in terms of file size, a variety of data, or other aspects.

In spite of data processing development, both approaches preserve their advantages and are orthogonal to each other. Sometimes it is advisable first to use non-parametric methods and then use a parametric method for the final estimate. In [9], we proposed a method that use parametric estimates to automatically detect performance degradations. However, it is highly dependent on finding and measuring a suitable independent variable. We believe that non-parametric modelling could be a better solution.

3.2 Regressogram

We introduce the regressogram [1, 16, 5] method, also called the binning approach, that is the simplest non-parametric estimator of a regression function. A regressogram is an estimator of the regression function that is constant by piece-wise, which can be thought of as approximating the data by a step function. This estimator is analogous to histograms for density estimation. We cover the observation space of X variable by disjoint buckets, and the estimated value of regressogram in a bucket is the mean of the Y-values for the X-values inside that bucket. For simplicity, we assume that the covariates X_i are from a distribution $0 \leq X \leq 1$. Similar to the histogram, we choose k as the number of buckets. Then we divide interval [0,1] into k buckets:

$$B_1 = [0, h],\ B_2 = [h, 2h],\ \ldots,\ B_k = [(k-1)h, kh]. \tag{3.3}$$

Let n_j denote the number of observations in bucket B_j. In other word $n_j = \sum_i I(X_i \in B_j)$ where:

$$I(X_i \in B_j) = \begin{cases} 1, & if\ X_i \in B_j \\ 0, & if\ X_i \notin B_j \end{cases} \tag{3.4}$$

Then, we define estimate $\bar{Y}_j$ as a mean of Y_i values in B_j:

$$\bar{Y}_j = \frac{1}{n_j} \sum_{X_i \in B_j} Y_i. \tag{3.5}$$

Finally, we define model $\hat{m}(x) = \bar{Y}_j$ for all $x \in B_j$:

$$\hat{m}(x) = \sum_{j=1}^{k} \bar{Y}_j I(x \in B_j) \tag{3.6}$$

The fitness of estimation of regressogram model depends primarily on the number of buckets into which we divide the interval with X-coordinates. There are simple plug-in methods to determine the optimal number of buckets, that have been designed primarily for density estimation with histogram and give good starting points for a number of buckets. Among the most used rules to choose bucket size belongs to Sturge's Rule [19], other alternate rules includes for example Scott's Rule [14], that will be introduce in Paragraph 3.4.1.

Sturge's Rule It is a widely used rule to choose bucket size, that works best for continuous data that is normally distributed and symmetrical. This rule should give the regressogram that represents the data well when the estimated data is not skewed. Its formula is follows: $K = 1 + 3.322 \log_N$, where the K is number of buckets and N is the number of observations in the data-set.

3.3 Moving Average

The moving average, also called rolling average or moving mean, is a widely used statistical estimator in technical analysis, that helps to smooth the effects of anomalies (such as outliers[2]) within data. The moving average methods belong to the group of delayed indicators of technical analysis, that analyse the sub-intervals of the same length. The basic idea is to create of average from the individual sub-intervals with Y-values continuously run through points on X-axis. Its primary objectives include smoothing out short-term fluctuations and highlight longer-term trends or cycles.

Any moving average method performs the analysis using a fixed width of the data window. The width of this window, so-called the averaging period, represents the length of an interval over which the moving average is computed. It is a crucial aspect of performing the analysis successfully. A narrow width removes the noise less effectively and leads to the identification of many false turning points[3]. On the contrary, a wide window width improves noise removal but increases the delay time in recognising the turning points.

There are two types of moving average — Centred Moving Average and Right-Aligned Moving Average — referring to the data window. The averaging data window consists of a centre and two halves at Centred (two-sided) Moving Average. If we denote by n the width of this window, then halves will have width k such that $n = 2k + 1$. The computation of the value of Centred Moving Average for a point X_i is given by:

$$MA_i^c(n) = \frac{X_{i-k} + \cdots + X_i + \cdots + X_{i+k}}{n} = \frac{1}{n} \sum_{j=-k}^{k} X_{t+j}. \tag{3.7}$$

It is used to identify turning points in the trend of past data. A Right-Aligned Moving Average is a lagged modification of the Centred Moving Average, and therefore it has the same smoothing properties. Generally, the value of this average at i^{th} position equals the value of Centred Moving Average at position $i - k$, where k denotes the half-width of the averaging window. Formally, this means the following identity:

$$MA_i^r(n) = MA_{i-k}^c(n), \tag{3.8}$$

where $MA_i^r(n)$ is value of Right-Aligned Moving Average computed as:

$$MA_i^r(n) = \frac{X_i + X_{i-1} + \cdots + X_{i-n+1}}{n} = \frac{1}{n} \sum_{j=0}^{n-1} X_{i-j}. \tag{3.9}$$

In next subsections we introduce two basic and commonly used moving averages: the Simple Moving Average (SMA), which is the average of Y-values over a defined number of subintervals, and the Exponential Moving Average (EMA), which gives greater weight to more recent points. Finally, we briefly introduce a Simple Moving Median (SMM), a modified version of SMA.

[2]Outliers are observation points that are distant from most other observations.

[3]The turning points change the direction in the data pattern.

3.3.1 Simple Moving Average

The Simple Moving Average (SMA) computes the arithmetic average of the previous n data points, or more frequently is the average taken from an equal number of data on either side of central value. The SMA with narrow window width is more volatile, but its output is closer to the source data. On the contrary, the wider the width of the window is, the smoother the resulting estimate. In summary, it means that the smoothing effect of this approach increases as the window width widen. The main limitation of this method is that with significant changes in the value in a window, the prediction of the model may not adapt well. These facts are the primary reasons why this method is recommended only for estimation with narrow window width and for data-sets that includes a horizontal pattern[4].

It is an unweighted moving average, so each data point has the same weight $\omega_i = 1$ ($\psi_i = \frac{1}{n}$). For the sake of completeness, we repeat how this method is computed:

$$SMA_i(n) = \frac{1}{n}\sum_{j=0}^{n-1} X_{i-j}, \tag{3.10}$$

which represents the same relation that was introduced in the Formula 3.9, where the SMA_i is estimation for point X_i, X_j is a current Y-value in the point X and n is the count of the values includes in the computation. When we use only past data points to computation, then it is advantageous in several application of this approach.

3.3.2 Exponential Moving Average

The Exponential Moving Average (EMA) as a type of weighted moving average method that applies weighting factors which decrease exponentially in the distance from currently estimated point. So, weights applied to data points do not have a linear character as in the case of SMA but instead they have an exponential character. The values close to the currently estimated point have the most influence on this point, and therefore these values have a higher weight than more distant points, whose weight exponentially decreases with increasing distance. EMA is computed as:

$$EMA_i(\lambda, n) = \frac{X_i + \lambda X_{i-1} + \lambda^2 X_{i-2} + \cdots + \lambda^{n-1} X_{i-n+1}}{1 + \lambda + \lambda^2 + \cdots + \lambda^{n-1}} = \frac{\sum_{j=0}^{n-1} \lambda^j X_{i-j}}{\sum_{j=0}^{n-1} \lambda^j}, \tag{3.11}$$

where

- X_i is the value of i^{th} point from data-set,
- n is the number of total points includes in window,
- $0 < \lambda \leq 1$ is a constant that determines the weight of a point.

We call λ the decay factor, which expresses the extent to which more distant points affects the final estimate. When $\lambda < 1$, the EMA assigns higher weights to the most recent points. By varying this value, one can adjust the weighting to give a higher or lesser weight to the most recent data points according to current requirements of the final estimate. The properties of the EMA are as follows:

$$\lim_{\lambda \to 1} EMA_i(\lambda, n) = SMA_i(n), \quad \lim_{\lambda \to 0} EMA_i(\lambda, n) = X_i \tag{3.12}$$

[4]In the horizontal pattern, data values fluctuate around their constant mean.

So, when λ approaches one, the value of EMA converges to the value of corresponding SMA. When λ approaches zero, then the value of EMA becomes the last data point [22].

The significant difference between EMA and SMA is the sensitivity for the variations in the data used in its computation. More specifically, the SMA assigns equal weights to all points, while the EMA assigns higher weights to nearest points. Both approaches are similar because they are using in the technical analysis used to smooth out fluctuations. The primary advantage of EMA is that is more reactive to the latest variations in data points, which makes it a more preferred approach than SMA.

3.3.3 Simple Moving Median

The Simple Moving Median (SMM) is similar to the SMA method, except that instead of computation of average values, it computes the median from the values in the window. A median is conceptually similar to an average but, the advantages of SMM is that it is not affected by outliers and therefore it is more robust estimator than SMA. The normal distribution of the fluctuations in the trend in data statistically ensures that the MA is an optimal estimator for recovering the underlying trend in estimated data-set of points.

However, the significant deviations from the trend do not have a high probability of occurrence in the normal distribution, and therefore they have a disproportionately large effect on the trend estimate. The SMM can be statistically optimal when the fluctuations are instead assumed to be Laplace distributed[5]. It tolerates significant changes in data better than SMA because the Laplace distribution places a higher probability on extraordinary points than does the normal distribution. When the SMM is central, the smoothing is identical to the median filter, which is known from image signal processing.

3.4 Kernel Regression

The Kernel Regression is a non-parametric technique that estimates the conditional expectation of a random variable. It builds on the primary idea of smoothing, which assumes the follows assertion: when the m is a smooth function, then the observations in points X_i (close to a point X) contains the information about the value m in the point X. It is, therefore, appropriate to use the local averages of data points close to the point X to obtain the estimate $m(x)$. So basically it finds a non-parametric relation between a pair of random variables X and Y. Generally, the kernel estimates of the regression function m in point X are defined as follows:

$$\hat{m}(x,h) = \sum_{i=1}^{n} W_i(x,h)Y_i, \tag{3.13}$$

where function $W_i, for\ i = 1, 2, \ldots, n$, are called weights independent on Y, but dependent on positive number h, that is called smoothing parameter. The type of W depends on kernel function K. All kinds of kernel estimates depend on this kernel function, and therefore we introduce its definition and properties:

Definition 3.4.1 *Let v, k be a non-negative integers, $0 \le v < k$, K is the kernel of k series with the class of all these functions denoted as S_{vk}, if K satisfies the following properties:*

[5]Laplace distribution, also called the double exponential distribution, is the distribution of differences between two independent variates with identical exponential distributions.

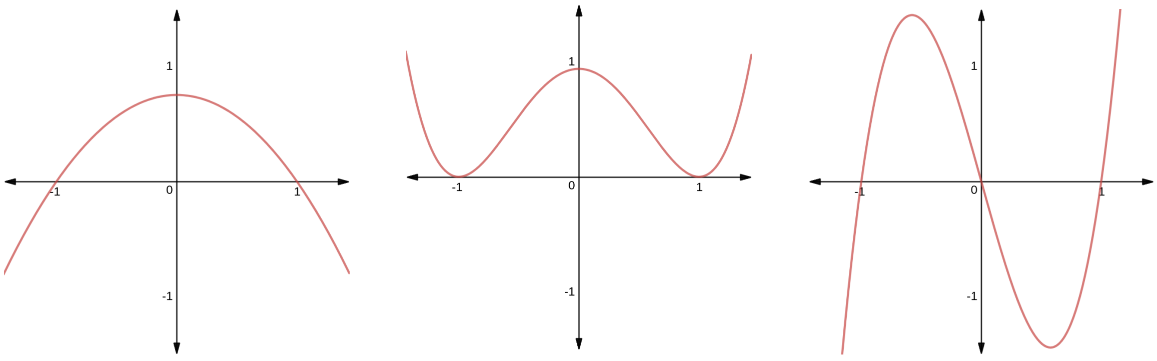
1. *K meets the Lipschitz Condition*[6] *on the interval* $[-1,1]$*, i.e.* $|K(x)-K(y)| \leq L|x-y|$ *for* $\forall x, y \in [-1,1], L > 0$,

2. *The support*$(K) = [-1,1]$*, i.e.* $K = 0$ *within the interval* $[-1,1]$,

3. *K meets the momentous conditions:*

$$\int_{-1}^{1} x^j K(x)dx = \begin{cases} 0 & 0 \leq j < k, j \neq v, \\ (-1)^v v! & j = v \end{cases} \tag{3.14}$$

and $\int_{-1}^{1} x^k K(x)dx \neq 0$*, these value marks* $\beta_k(K)$.



(a) $K(X) = \frac{3}{4}(1-X^2)$ **(b)** $K(X) = \frac{15}{16}(1-X^2)^2$ **(c)** $K(X) = -\frac{15}{4}X(1-X^2)$

Figure 3.1: The examples of kernels on the interval [*−1*, *1*]: (a) Epanechnikov, (b) Quartic and (c) kernel from class S_{13}.

There are many types of kernel estimates of the regression function, which are asymptotically equivalent[7] and therefore their selection is not essential from this point of view. We can list, e.g., the Nadaraya-Watson estimator as the most useful one, or exists some alternative kernel estimators, such as Priestley and Chao or Gasser and Muller. We will describe the Nadaraya-Watson estimator and its construction, and also we will illustrate the influence of the smoothing parameter h on the quality of the resulting estimate. We can define $K \in S_{0k}$, where k is the even number and let $K_h(\cdot) = \frac{1}{h}K(\frac{\cdot}{h})$. For given point X, $h < X < 1 - h$ are the weights of Nadaraya-Watson estimate given by the formula:

$$W_i(X,h) = \frac{K_h(X - X_i)}{\sum_{j=1}^{n} K_h(X - X_j)}, \quad \sum_{j=1}^{n} W_j(X,h) = 1. \tag{3.15}$$

Note that a kernel estimate is not defined for $\sum_{i=1}^{n} K_h(X - X_i) = 0$, therefore, in the case of $\frac{0}{0}$, we put $\hat{m}_{NW}(X,h) = 0$, where $\hat{m}_{NW}$ represent defined type of kernel estimate. When we limit to the estimations of the function m only in the points X_i, $i = 1, \ldots, n$, then for $h \to 0$ is apply:

$$\hat{m}_{NW}(X_i,h) \to \frac{K(0)Y_i}{K(0)} = Y_i. \tag{3.16}$$

[6] https://www.encyclopediaofmath.org/index.php/Lipschitz_condition

[7] Two kernel estimates K_1 and K_2 are asymptotically equivalent if limit $\lim_{x\to\infty} \frac{k_1(x)}{k_2(x)}$ exists and is equal to 1.

This means that with a narrow width of smoothing window the estimate reproduces the data. On the contrary, the wide width of the smoothing window leads to overlying, and so to an average of the data. Formally said, for $h \to \infty$ is apply:

$$\hat{m}_{NW}(X_i, h) \to \frac{\sum_{j=1}^{n} K(0) Y_j}{\sum_{j=1}^{n} K(0)} = \frac{K(0) \sum_{j=1}^{n} Y_j}{nK(0)} = \frac{1}{n} \sum_{j=1}^{n} Y_j. \tag{3.17}$$

The Graph 3.2 illustrates the construction of the estimates in the point X_0, that is composed of the five observations $(X_1, Y_1), \ldots, (X_5, Y_5)$, that are represents by black crosses. The red parabola represents the Epanechnikov kernel K_h and the blue circles views the values of weights $W_i = \frac{K_h(X_0 - X_1)}{\sum_{i=1}^{5} K_h(X - X_i)} = 0$ for $i = 1, \ldots, 5$. The resulting estimate of the regression function $\hat{m}$ in the point X_0 is represents as the blue cross.

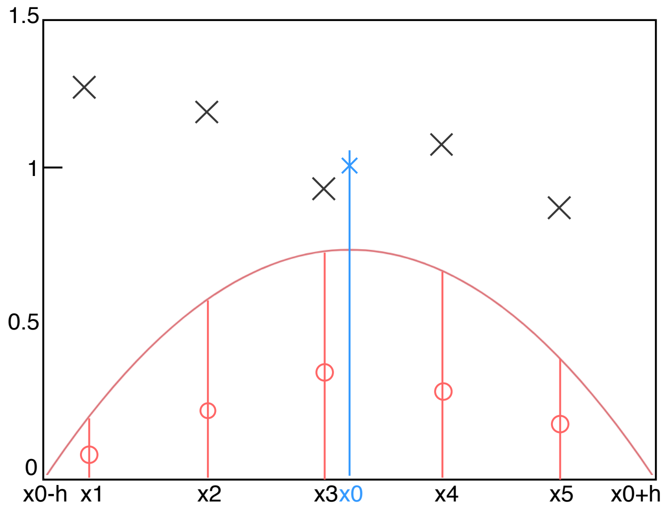


Figure 3.2: Illustration of Nadaraya-Watson kernel estimate in the point X_0 with use of Epanechnikov Kernel, that is shown in the Figure 3.1a.

Generally, the kernel regression performs the most suitable estimates of the regression functions, and therefore it is used frequently in technical analysis. The most critical factor is the right choice of smoothing window width (smoothing parameter h), and therefore we will discuss it in the following sections. It should be noted, that the final decision on the estimated curve is partially subjective, as the asymptotically optimised kernel estimates contain quite a large amount of the noise. Definitions and formulae in this section are mostly inspired by books and online references, which can be found in [6, 21].

3.4.1 Smoothing Parameter Selection

Choosing the right value of smoothing parameter is most critical factor of Kernel Regression. The selection of the width of the smoothing window significantly affects the fitness of the resulting estimate — the wide window width leads to overlay (so to average data), and a narrow window width to underlay. We will describe a selection of bandwidth selectors in the context of this work. Definitions and Formulae in the following subsections are mostly inspired by [12].

Cross-Validation. The earliest, fully automatic and consistent bandwidth selectors were those based on cross-validation. This idea is probably the simplest and most widely for estimating prediction error[8]. This method directly estimates the expected out-of-sample error, also called average generalisation error, when the function $\hat{m}(x)$ is applied to an independent test sample from the joint distribution of X and Y. There are multiple ways to perform

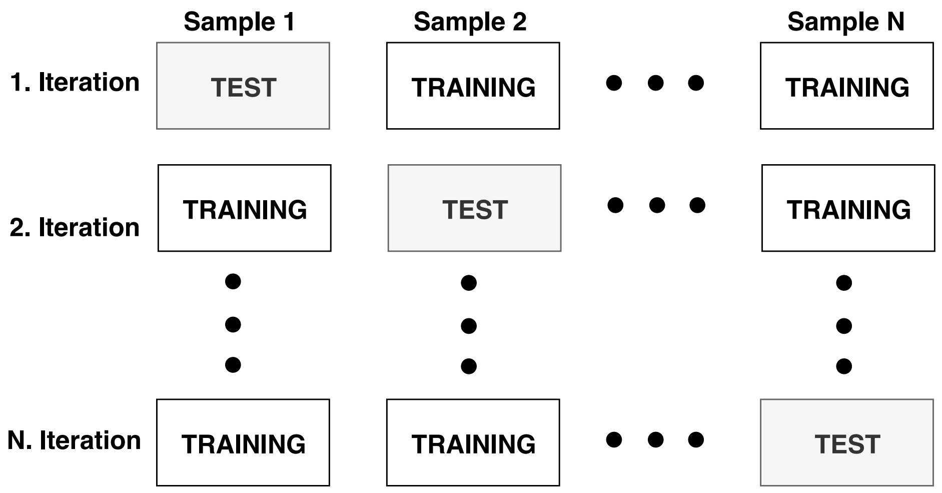


Figure 3.3: Leave-One-Out Cross-Validation on data-set with N samples.

this approach, such as K-Fold Cross-Validation or Exhaustive Cross-Validation, but instead, we focus on Least-Squares Cross-Validation (LSCV) with Leave-One-Out estimator (see Figure 3.3), that is based on minimising:

$$LSCV(h) = n^{-1} \sum_{i=1}^{n} (Y_i - \hat{g}_{-i}(X_i))^2, \tag{3.18}$$

where $\hat{g}_{-i}(X_i)$ is the estimator of $m(X_i)$ formed by leaving out the i^{th} observation when generating the prediction for observation i:

$$\hat{g}_{-i}(X_i, h) = \sum_{\substack{l=1 \\ l \neq i}}^{n} W_l(X_i, h) Y_l, i = 1, \ldots, n. \tag{3.19}$$

Akaike Information Criterion. AIC-based method of Hurvich et al. [3] is based on minimising a modified Akaike Information Criterion (AIC). It is a statistical technique based on in-sample fit to estimate the probability of a model to estimate the processed data-set. The model that minimises AIC the best in comparison with other models is chosen as a final model for the resulting estimate. So, this approach is based on the minimisation of the criterion:

$$AIC_C = \ln \hat{\sigma}^2 + \frac{1 + \frac{tr(H)}{n}}{1 - \frac{tr(H)+2}{n}}, \tag{3.20}$$

where

$$\hat{\sigma}^2 = \frac{1}{n} \sum_{i=1}^{n} (Y_i - \hat{g}(X_i))^2 = \frac{Y'(I-H)'(I-H)Y}{n} \tag{3.21}$$

[8]The prediction error is the failure of estimated points in comparison to its expected value.

where $\hat{g}(X_i)$ represents a non-parametric estimator and H represents the matrix of kernel weights about the size $n \times n$ (i.e. the weighting function) with its $(i,j)^{th}$ element given by $H_{ij} = \frac{K_h(X_i,X_j)}{\sum_{l=1}^{n} K_h(X_i,X_l)}$, where $K_h(\cdot)$ is a generalized product kernel. Both described methods (LSCV and AIC) are asymptotically equivalent, which means that the kernel estimates with a window width computed using these methods achieve approximately the same fitness.

Scott's Rule of Thumb. It is a high-speed computation for determining the smoothing bandwidth for kernel regression. Initially, this rule was designed for density estimation but it is usable for kernel regression too. Typically it produces a wider bandwidth and therefore it is useful to estimate a gradual trend, $bw = 1.059 * a \ * n^{-\frac{1}{5}}$, where $A = \min(\sigma(x), \frac{IQR(x)}{1.349})$, where $\sigma(x)$ represents the Standard Deviation and IQR represents the Interquartile Range[9].

Silverman's Rule of Thumb. This rule belongs to the most popular methods which use the rule of thumb. Since this rule was originally designed for density estimation, it uses the normal density as a prior for approximating. For the necessary estimation of the $\sigma(x)$ it proposes a robust version of making use of the Interquartile Range. If the true density is uni-modal, fairly symmetric and does not have fat tails, it works fine. Its formula is follows: $bw = 0.9 * a \ * n^{-\frac{1}{5}}$, where A is the same as above.

[9]The IQR is a measure of variability, based on dividing a data set into quartiles.

Chapter 4

Non-Parametric Post-Processors

This chapter introduces the design and implementation of the post-processors implements the non-parametric modelling techniques (*regressogram*, *moving-average* and *kernel-regression*) as introduced in Sections 3.2, 3.3 and 3.4. Each of these post-processors was designed as an independent component with its own interface within the Perun framework. Their main task is to process the raw performance data (obtained by collectors introduced in Section 2.2) for further interpretation, mainly for detection methods or visualisers. In the next sections, we discuss the individual post-processors, their possibilities and benefits for the process of automatic detection of performance changes or manual interpretation.

4.1 Requirements

We begin with requirements on the resulting post-processors, which must be taken into mind at design and implementation. However, not all requirements can be met without reservations, primarily because of the trade-off between speed and accuracy (i.e. the best fitness of the resulting estimate). The requirements are based on discussions with Perun collaborators.

1. **Flexibility.** The post-processor should provide as many options as possible to set the different values of parameters to perform estimate by the user.

2. **Automation.** The average user does not know the appropriate values for parameters, and therefore the post-processor should also perform the analysis that can achieve the acceptable results without the manual user involvement. The default values of significant parameters should be set to achieve the acceptable fitting estimates.

3. **Independence.** The goal is to minimise dependencies of post-processors on third-parties packages or other possible dependencies. More specifically, the minimal proliferation of new dependencies is needed within the framework Perun.

4. **Efficiency.** The post-processor should perform the estimation effectively so that the overall run-time of Perun batch jobs is not significantly prolonged. At the same time, the efficiency should not be achieved on great expenses on the fitness of estimate, which strongly affects the results of the next phases of the whole process of automatic detection.

4.2 Regressogram

In this section, we will describe first non-parametric post-processor that implements the statistical modelling technique called regressogram. This approach follows the simple modelling idea that was described in Section 3.2. The following sections describe how this post-processor implements this idea and how it is beneficial and innovating in our framework.

Analysis. The implementation of regressogram method is intuitive, yet there are several ways how it can be implemented. We will first explore the available options. Multiple third-party packages provide built-in methods to implement regressogram: in particular `numpy.digitize`, `numpy.histogram` or `scipy.stats.binned_statistic`. However, the first two packages require additional manipulation with processed data-set or do not provide a broad range of options in comparison to the last package, which has the expected functionality and therefore was selected from these available methods.

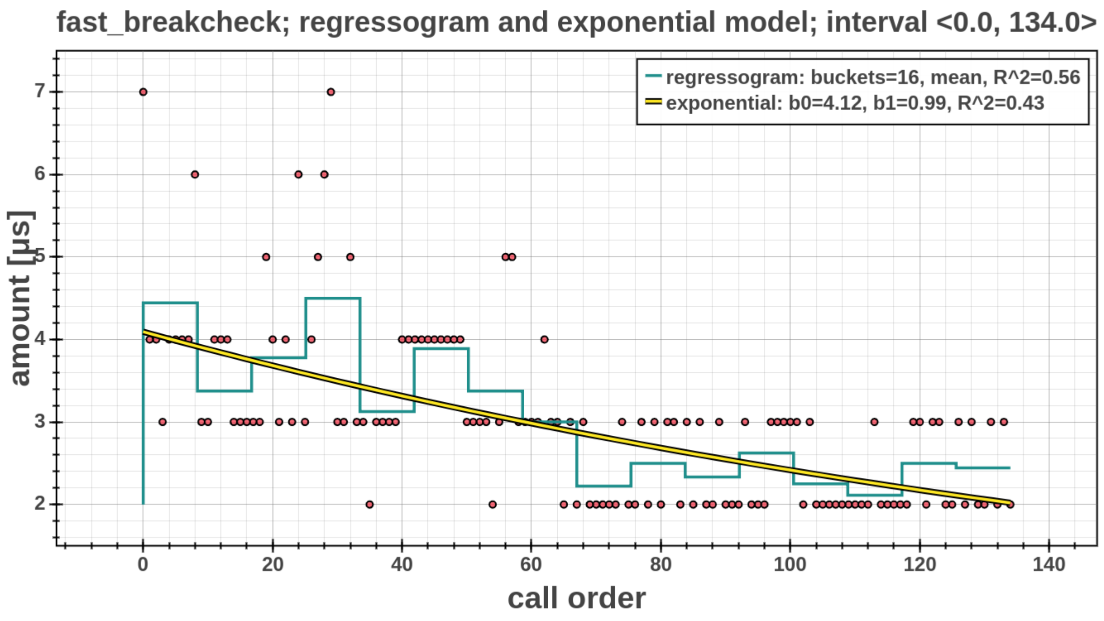


Figure 4.1: The example of the models created on the raw performance data collected by trace collector on Vim binary: model shown in cyan represents the regressogram model, and model shown in yellow the best regression model created by regression-analysis post-processor. The regressogram model was created with default value of parameters: the count of buckets was determined by `Doane's` method and in each bucket we computed the average of values. This figure was created by extended `scatter` module from `view` unit (see Figure 2.1), specifically was used the method `render_step` from `methods` module of regressogram post-processor. On the x-axis is the call number in the collect record (call order), and on the y-axis is the run-time (in microseconds) of function in the specific call (amount).

`scipy.stats.binned_statistic`[1]**.** This method, from the `SciPy` (Science Python) [4] package, performs the computation of the binning approach (regressogram) for a given data-set (see Listing 1). This method allows the computation of the different aggregation

[1]https://docs.scipy.org/doc/scipy-0.16.0/reference/generated/scipy.stats.binned_statistic.html

function (such as mean, median, sum, etc.) from the values within each bucket. Besides, it allows one to define custom aggregation function to compute the resulting values within buckets, which shows the possibility of the natural expansion of this post-processor.

Implementation. The implementation of this post-processor is composed of the two primary modules: `run` and `methods`. The first module `run` covers its main functionality and implements the interface within Perun. The output of this module is the modified profile with the results of analysis performed by this post-processor, i.e. regressogram models. The module `methods` is a public interface of implemented non-parametric technique, `i.e. regressogram`. It contains the method for computation of the specific regressogram models and implements helper methods used for their manipulation (e.g. to render these models such as in the Figure 4.1).

```
from scipy import stats

x = [1, 2, 3, 4, 5, 6]
y = [1, 1, 1, 1, 1, 1]

>>> stats.binned_statistic(x, y, statistic='mean', bins=3)
BinnedStatisticResult(
    statistic=array([1, 2, 3]),
    bin_edges=array([1, 2.667, 4.334, 6]),
    binnumber=array([1, 1, 2, 2, 3, 3])
)
```

Listing 1: The demonstrations of calculation of regressogram model from samples x and y. In the resulting structure `BinnedStatisticResult` we can see the `statistics`, that contains the computed values of the model, i.e. the means of each bin. Another important item is `bin_edges`, that contains the values for rendering this model.

4.3 Moving Average

This section describes the design and implementation of the post-processors that implements moving-average methods. This post-processor implements the three methods of this approach: *Simple Moving Average* (SMA), *Exponential Moving Average* (EMA) and *Simple Moving Median* (SMM) as the simple modification of these two methods. The theoretical background of all these methods was introduce in Section 3.3.

4.3.1 Analysis

The moving averages methods are included in many packages. Table 4.1 shows the comparison of methods, which we considered from the different aspect, such as dependency, adjustability and performance. From these methods, the best results in the run-time achieved the method `MA` from `Ta-Lib` package. However, this is an problematic package, because during installation there may occur multiple issues that are reported on the project repository[2]. Choosing this package would then contradict the requirements of flexibility and potentially cause issues in Perun's continuous integration system.

Table 4.1: The table shows the run-times of the individual tested modules, that contains methods for computing the SMA. The methods `cumsum`, `convolve` and `fftconvolve` (in table `fft`) are from the `NumPy` (Numerical Python) package, methods `Series.rolling` and `rolling_mean` (in table `rolling`) are from the `Pandas` package, and `Ta-Lib` is the Technical Analysis Library, that contains a method called `MA`. We tested these on the array containing *1 000 000* samples, and the width of the window was set to *100*. The measured metrics (mean, max and min) were calculated on the *50* repeats of the calculations of all methods. The green colour represents the best run-times in comparison to the remaining methods, and the red colour represents the slowest run-times. The chosen method is highlighted with the yellow colour.

	cumsum	**talib**	**fft**	**convolve**	**pd.Series**	**pd.rolling**
mean	0.00816s	0.00408s	0.09467s	0.03339s	0.03031s	0.03004s
max	0.03334s	0.06364s	0.17439s	0.05521s	0.12372s	0.04364s
min	0.00715s	0.00265s	0.08243s	0.03131s	0.02574s	0.02631s

The adjustability of different options is one of the significant factors, that influenced our selection. The options such as *window-type* (such as hamming, triangular or boxcar), *minimal count* of samples in the window or variability of *centre*, i.e. whether the window is *right-aligned* or not, are the critical aspects in performing the analysis and in the fitness of its resulting models. The second fastest method (`cumsum`) does not contain any of the listed options, as it requires the additional steps (e.g. calculation of aggregation function within windows) after the calculation of cumulative sum[3] by original method. After considering all critical factors, we decided to choose `rolling` method from `Series` class of `Pandas` package. This method achieved the average values of run-time in all measured aspects and provides the multiple options to calculate rolling window. The `Series` class also allows calculating of EMA and SMM, which concludes our requirements for this post-processor.

4.3.2 Implementation

The moving average post-processor is composed of two modules: `run` and `methods`. The first module implements the commands and options for Command Line Interface. This post-processor implements three different types of moving average methods that were designed as three separate commands with the same names as the methods: *SMA*, *SMM* and *EMA*. The interface was designed concerning the requirements of this post-processor and therefore is composed of common options for each command, the individual commands and specific individual options for these commands. The interface is easy to expand with the new moving average methods or the new options for all these methods concerning the changes in the entire Perun framework.

The module `methods` implements the computational logic of the whole post-processing phase. The following methods perform the computation of individual moving average methods itself: `pandas.DataFrame.rolling` for *SMA* and *SMM*, and `pandas.DataFrame.ewm` for *EMA*. Figure 4.2 shows the example models created by this post-processor, where we

[2]https://github.com/mrjbq7/ta-lib
[3]A cumulative sum is a sequence of partial sums of a given sequence.

can see all these moving-average methods. The most critical aspect of this computation is the choice of window width. We have proposed a heuristic (see Algorithm 1), which estimates the window width iteratively (unless the model achieves the desired fitness). This heuristic is used by default, when the window width is not specified implicitly by user, so one can obtain models that adequately fit given data-set without broader knowledge about the actual data.

```
Method iterative_computation($x$, $y$)
    $R^2 = 0.0$
    $\square$ = max($1$, $(x_{max} - x_{min}) * \xi_{IL}$)
    while $R^2 < R^2_{min}$ and $\Delta\square$ do
        $\mathcal{M}$, $R^2$ = MovingAverage($y$, $\square$)
        $\Delta\square$ = $\square$ - ComputeWindowChange($\square$, $R^2$)
        $\square$ = ComputeWindowChange($\square$, $R^2$)
    return $\mathcal{M}$, $R^2$, $\square$
```

Algorithm 1: The heuristic for the computation of moving average models iteratively. The value of the coefficient of determination R^2 represents the fitness of the model. The value of window width ($\square$) is set to a certain percentage of the total length of the x-interval (denoted as ξ_{IL}). When this value is less than 1, then a width of the window is set to 1. Subsequently, the iteration is repeated, unless the value of the coefficient of determination is not high enough and unless the window width varies. Each cycle calculates the new moving average model with the current value of window width and adjusts the new value of window width for the next iteration. The change of this value depends on the current window width and coefficient of determination.

However, the computation of the EMA does not use the known window width, but instead it uses the specific decay parameter, which has the role of smoothing factor. This factor λ $(0 < \lambda \leq 1)$ (also called the smoothing constant) is essentially a weight applied to the points in closest surroundings. EMA is also sometimes specified using the *span*, *centre of mass* or *half-life* parameters, then the λ parameter is related to them as:

$$\lambda = \begin{cases} \frac{2}{s+1}, & for\,\text{span}\, s \geq 1 \\ \frac{1}{1+c}, & for\,\text{centre of mass}\, c \geq 0 \\ 1 - \exp^{\frac{\log 0.5}{h}}, & for\,\text{half-life}\, h > 0 \end{cases} \tag{4.1}$$

The *span* corresponds to what is commonly called an *N-point* EWMA, the *center of mass* (com) has a more physical interpretation and can be thought of in terms of span: *com* = (*span* − 1)/2, and *half-life* is the length of the interval for the exponential weight to reduce to one half. The validation for these parameters was implemented in the module *methods.py* and subsequently is passed to the computational method `pandas.DataFrame.ewm`, which uses them to perform the analysis [7].

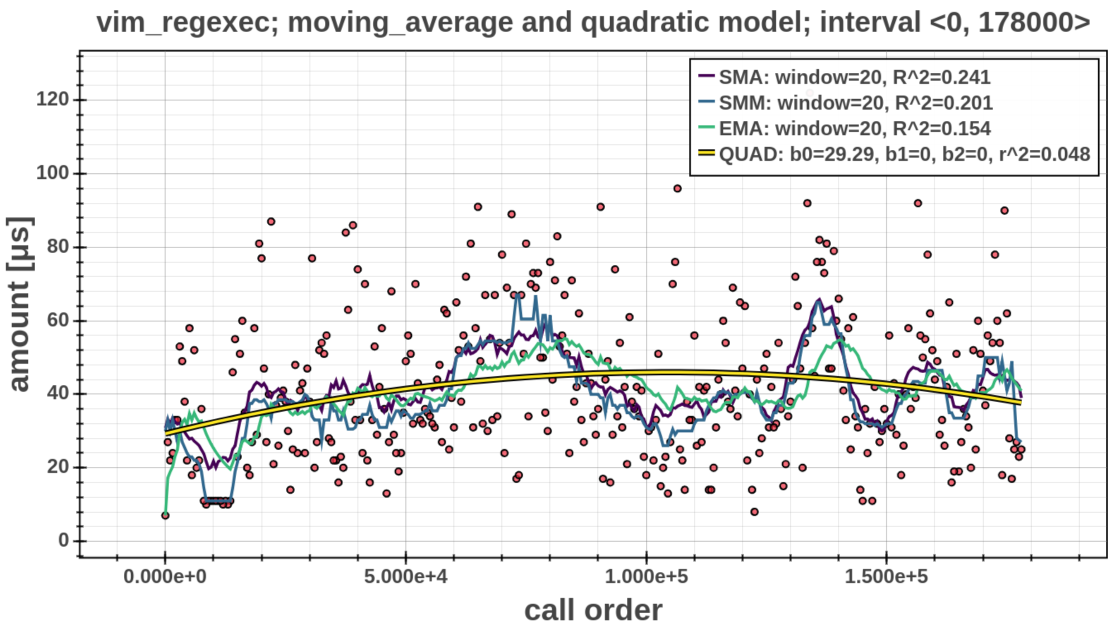


Figure 4.2: The example of moving average models that were created by the moving-average post-processor and the best regression model created by regression-analysis post-processor. The data-set for the analysis was collected by the `trace` collector on the VIM binary. It was performed with the window width equal to *20* at all types of moving average methods (SMA, SMM and EMA), and the count of the minimal samples in the window was set to *1*, to achieve better rendering. The viewer unit of the PERUN generated the scatter plot. Specifically, it uses the method to render non-parametric models, that was implemented with these post-processors. Though the difference in the SMA and EMA seems apparent (almost 0.1 difference in R^2), either one cannot be said to be better than one other. The SMA is usually more appropriate for a longer movement at the overall interval, which confirms his best coefficient of determination R^2 in this estimate. The SMM has reached the mean value in comparison to SMA and EMA because it is susceptible to rare points (outliers), which have not been present in this data-set. As we can see, all three approaches of moving-average have achieved better estimate (higher value of the coefficient of determination R^2) in comparison to regression quadratic model, as the representative of the best regression models.

4.4 Kernel Regression

The last post-processor implements the kernel regression methods, which (in theory) was introduced in Section 3.4. This chapter introduces the analysis of existing solutions that are available in the different statistics packages and lists their advantages concerning our requirements. Subsequently, we will introduce how the chosen implementations were integrated within PERUN.

4.4.1 Analysis of Requirements

We had set two requirements that should be met to achieve a successful analysis.

Kernel. The kernel is a weighting function, which significantly affects the resulting estimate because its centre is placed right over each data point. Since it is necessary to process various data distributions, it is appropriate to support different types of kernels. It is advised to choose kernels that are continuous on the whole definition domain, so then the estimated regression function inherits the smoothness of the kernel. For example, the *Gaussian* kernel is less steep, and hence the resulting kernel model reflects the higher number of neighbourhood points, which weight decreases from the centre of the kernel. On the other hand, the *Tricube* and *Epanechnikov* kernels put more emphasis on the currently estimated point, and therefore they reflect the smaller number of neighbourhood points, which have bigger weights.

Bandwidth. The value of bandwidth (i.e. smoothing window width or smoothing parameter) significantly affects the smoothness or roughness of the kernel estimate, because it controls how wide the probability mass is spread around the currently estimated point. The ignorance of data distribution bears the danger of under-smoothing or over-smoothing. It is therefore important to provide users with the different methods that automatically select the suitable bandwidth, such as simple (Scott's Rule of Thumb) or advanced (Cross-Validation) selection rules.

4.4.2 Analysis of Selected Implementations

Kernel-based methods are most popular non-parametric estimators, and there are many existing implementations of these techniques. Almost every package that provides statistical techniques contain some variations of kernel regression methods. From them we selected packages that meet our generic requirements on the post-processor (see Section 4.1) and requirements presented in the section above. In the end, we selected these suitable packages and implemented them all in our post-processors. Table 4.2 shows the comparison of the selected packages with individually available options to perform the kernel regression. In the following, we will introduce the individual packages (specifically their methods) that this post-processors uses to perform the kernel regression.

StatsModels (Statistics in Python) ([15]). This module provides classes and methods to performs different statistical techniques, in particular it provides non-parametric kernel regression class `KernelReg` from the module `nonparametric.kernel_regression`. This class provide a Nadaraya-Watson kernel regression (see Equation 3.15), also known as the local-constant type of regression. Moreover, it provides two different methods for automatic bandwidth selection: Least-Squares Cross-Validation (see Section 3.4.1) and AIC Hurvich bandwidth estimation (see Section 3.4.1), which meets our requirement about advanced selection rules. These automatic bandwidth selectors provide an efficient bandwidth estimation performed by `EstimatorSettings` object from the `nonparametric.kernel_density` module. This object allows one to specify the several properties that relate to how kernel bandwidth is estimated. Besides these advanced methods, `StatsModels` provides Scott's (see Section 3.4.1), and Silverman's (see Section 3.4.1) Rules for a simple estimate of the kernel bandwidth.

PyQt-Fit ([13]). This package provides both parametric and non-parametric regression techniques, e.g. kernel density estimation or linear regression methods. We can use the class `NonParamRegression` from the `pyqt_fit.nonparam_regression` module, that performs kernel-based non-parametric regression. The primary advantage of this class is that one can choose different types of kernels, that are provided by module `pyqt_fit.kernels` which meets our first requirement from Section 4.4.2. In addition to kernels available in this module, it is possible to implement a custom kernel type according to a prescribed class template. A `NonParamRegression` class performs a Nadaraya-Watson (see Equation 3.15) regression on the data using a selected kernel, but it also implements another non-parametric regressions using kernel methods (such as `Local-Polynomial` or `Local-Linear`) from the `pyqt_fit.npr_methods` module. The simplest way to specify kernel bandwidth is by user but often it is appropriate to use a pre-defined method. This package provides the Scott's (see Section 3.4.1), and Silverman's (see Section 3.4.1) Rules, which compute the bandwidth of the kernel automatically. At last, it provides an option to define own method to compute optimal kernel bandwidth.

Scikit-learn (Machine Learning in Python) ([10]). This package provides simple and efficient tools for data mining and data analysis. We can use the `model-selection` unit of this package, specifically its module named `metrics`, that contains method `pairwise_kernels`. This method computes the kernel between arrays X and optional array Y, where the kernel is represented as *Gaussian* kernel[4]. We use two base classes from this package and we have derived our own class from them, that will be describe in next Sections: `BaseEstimator` a class for all estimators and `RegressorMixin` as base class for all regression estimators.

	StatsModels	**PyQt-Fit**	**Scikit-learn**
Nadaraya-Watson	✔	✔	✔
Local-Linear	✔	✔	✘
Cross-Validation	✔	✘	✔
AIC-Hurvich	✔	✘	✘
Scott/Silverman	✔	✔	✘
Kernel Types	✘	✔	✘

Table 4.2: The comparison of the selected packages to perform the kernel regression. The rows show the individually beneficial options in kernel regression and cells denotes their presence in the selected packages. We focus on the primary properties of kernel estimation, such as the kernel bandwidth (smoothing parameter) or the kernel type.

4.4.3 Implementation

The implementation of this post-processor was again divided into two main modules: `run` and `methods`. The `run` module implements the interface in Perun and the second module contains the primary computation logic of this post-processor. The interface consists of three different modes of kernel-regression: *estimator-settings*, *kernel-smoothing* and *kernel-ridge*. These modes do not differ in the resulting kernel estimate, but they differ in the computation of the estimate. The purpose of these modes is to provide flexibility in constructing the kernel estimate. Moreover, the user has the option to choose from available kernel types, kernel smoothing methods and automatic selectors for optimal kernel bandwidth.

[4]The Gaussian kernel is the physical equivalent of the mathematical point.

Figure 4.3 shows the examples of kernel models, which was created by *estimator-settings* and *kernel-smoothing* modes.

The `kernel-smoothing` mode uses the implementation of `PyQt-Fit` and the `estimator-settings` mode uses the method from `StatsModels` package. This mode provides a few types of kernels, concretely *Gaussian*, *Tricube*, *Epanechnikov* and two kernels of the higher order, *Gaussian* and *Epanechnikov* of fourth order. As have been mentioned in the section above, both modes provide the Nadaraya-Watson kernel regression, and both provide the different options to an effective estimate of kernel bandwidth.

The last mode of this post-processor, named `Kernel-Ridge`, is partly built on the methods and classes from the `Scikit-learn` package, extended with our heuristics. The class `KernelRidge` implements the Nadaraya-Watson regression with the support of automatic kernel bandwidth selection. This automatic selection is performed by leave-one-out cross-validation (see Section 3.4.1), where we selected the specific value of kernel bandwidth from the given range, based on the minimising *Mean-Squared-Error*. The value of kernel bandwidth has a different meaning in this regression as usually. We compute the *Gaussian* kernel on the given data-set according to the formula of `rbf_kernel`: $K(x, y) = \exp\left(-\gamma|x - y|^2\right)$, for each pair of points (x, y) from the given data-set, where the γ represents the kernel bandwidth.

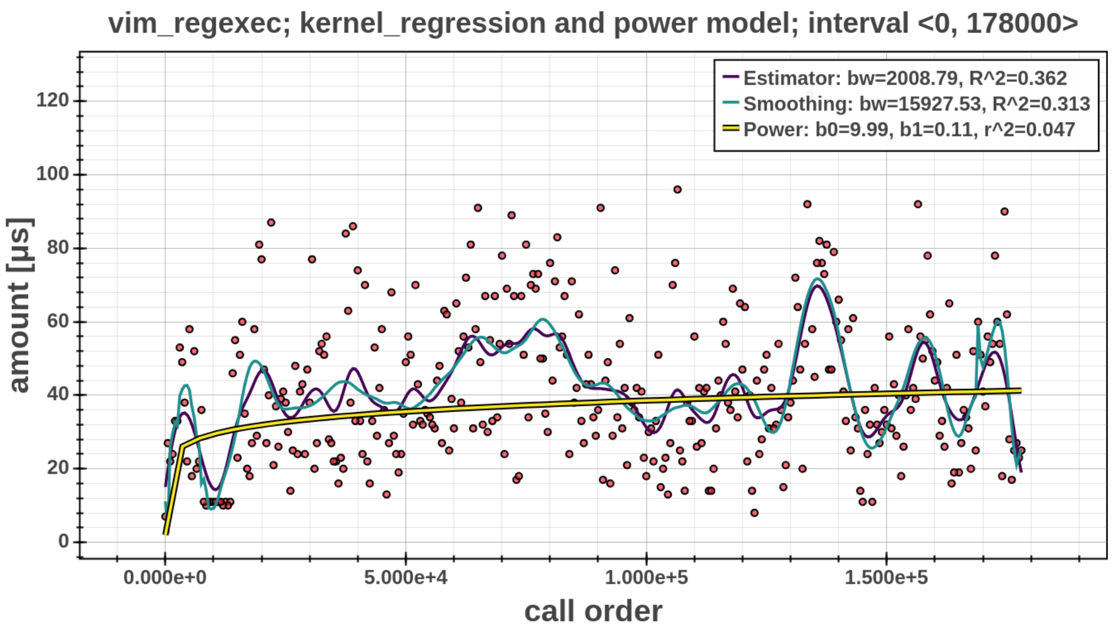


Figure 4.3: The figure shows two kernel models, that were created by this post-processor, specifically using `estimator-settings` and `kernel-smoothing` modes, and the best parametric model created by regression-analysis post-processor. The first model (shown in purple) has the kernel bandwidth estimated by `Estimator-Settings` object with the default values of its parameters. The second model (shown in cyan) was created with *Nadaraya-Watson* kernel regression method, where was used the *Epanechnikov* kernel and the bandwidth was determined with *Scott's Rule of Thumb*. As shown the graph legend, the kernel regression achieved a more fitting estimates with a markedly higher value of the coefficient of determination R^2 compared to the best parametric model (power model shown in yellow). This graph was again generated by PERUN view unit, which used the implemented method to render the kernel models.

Chapter 5

Detection of Performance Changes

This chapter introduces a design and implementation of methods for automatic detection of performance changes, mainly focused on models introduced in Chapter 4. Perun uses these detection methods as the last step in the whole process of performance detection preceded by collection and modelling of the performance data. We introduce two new implemented detection methods: *Integral Comparison* and *Local Statistics*. Initially, these methods were designed primarily for non-parametric models, however, they support all kinds of models in the Perun. Section 2.4 introduced the general description of the automatic detection of performance changes. We repeat, that the input of detection methods are two profiles (baseline and target) and the output is represented by `Degradation Info` structure (see Section 2.4 for more details).

The final phase in the evaluation of the performance changes is the classification of computed change indicators. The performance change indicators are metrics, statistics, or other values that can indicate that some change in performance could happen (e.g. average of model). The individual detection methods compute their own change indicators in a specific way according to their particular strategies. We can then compute the relative or absolute error from these indicators of baseline and target models. As shown in Figure 5.1, these errors then decide whether the performance changes will be detected or not, therefore the comparison of the individual errors and thresholds has a crucial role. Algorithm 2 shows the generic principle of comparing the errors. This algorithm takes the values of errors and thresholds, that are participating in comparison, as own parameters.

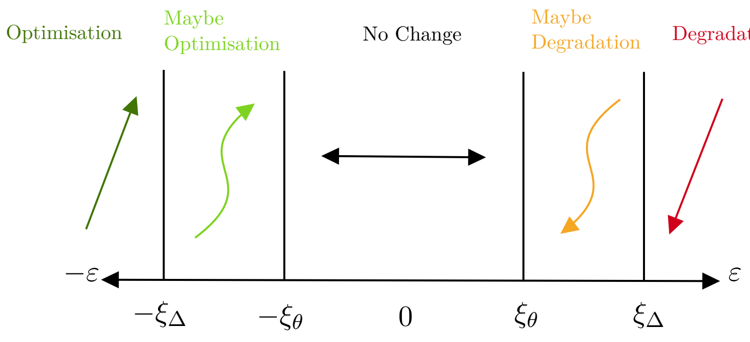


Figure 5.1: The illustration of classification the values of computed error ε by Algorithm 2. This algorithm compares these errors with given thresholds ξ_θ, ξ_Δ and based on the shown intervals determines whether change occurred or not.

```
Method ClassifyChange($\varepsilon$, $\xi_\theta$, $\xi_\Delta$)
    if $|\varepsilon| \leq \xi_\theta$ then
        $\Delta = \longleftrightarrow_{nochg}$
    else if $|\varepsilon| \leq \xi_\Delta$ then
        $\Delta = \nearrow_{mayopt}$ if $\varepsilon < 0$ else $\swarrow_{maydeg}$
    else
        $\Delta = \nearrow_{opt}$ if $\varepsilon < 0$ else $\swarrow_{deg}$
    end
    yield $\Delta$
```

Algorithm 2: The generic algorithm used to classify arbitrary computed errors ε (such as the relative error ε_{rel}) for detection of performance changes. The comparison of input parameters is the basis and according for determining the resulting performance change Δ. The threshold ξ_θ represents the acceptable interval to detect `NoChange` ($\longleftrightarrow_{nochg}$) and range between this threshold and threshold ξ_Δ represents detection of uncertain types: `MaybeOptimization` ($\nearrow_{mayopt}$) or `MaybeDegradation` ($\swarrow_{maydeg}$).

5.1 Integral Comparison

The first heuristic is based on the assumption that the areas under the curves that represent the individual models in graphs, should be approximately equal when no change has occurred. The main idea of this approach is to compute the definite integral under the given curves of baseline and target models. We designed this method for all kinds of models, i.e. both for the regression models created by `Regression-Analysis` post-processor (see 2.3) and for the models, that we described in the previous Chapter 4. By its complexity, the method brings an entirely new approach in the automatic detection of performance changes with the potential to more precise results.

5.1.1 Example

We will first illustrate the integral method on short example.

Integration of Parametric Models. The `Regression-Analysis` post-processor creates models that are represented by the set of coefficients (see Formulae 2.1 and 2.2). Therefore, we can compute the integral of these models from the function of one variable and these coefficients represent this function. For example, we can demonstrate the integral computation for the regression model, that is represented by function $y = x^2$ along the interval $[0, 10]$:

$$I = \int_0^{10} x^2 dx. \tag{5.1}$$

We use the `integrate`[1] module from the `SciPy` [4] package, that contains several functions to approximate definite integrals and numerically solve the differential equations. In particular, we use the `quad` method from this module to compute a definite integral of these models.

Integration of Non-Parametric Models The non-parametric post-processors create models that are represented by a set of samples (values of the model). Therefore, we need to

[1] https://docs.scipy.org/doc/scipy/reference/tutorial/integrate.html

use methods that compute a numerical approximation of integral from the set of samples. We decided to use the *Simpson's Rule* [2], that is a numerical method for computation such integral defined on the interval $[x_{start},\, x_{end}]$. This interval is usually broken into N equal-sized sub-intervals. However, the non-parametric models are given by the fixed set of samples, and therefore the N is equal to the length of this set. The values of the model within interval $[x_{start},\, x_{end}]$ we denote as x_i, for $i = 0, \dots, N$. When we suppose, that this model approximates the function $f(x)$, then the numerical computation of integral using Simpson's Rule looks as following:

$$\int_{x_{start}}^{x_{end}} f(x)dx \approx \frac{x_{end} - x_{start}}{3N}(x_0 + 4x_1 + 2x_2 + 4x_3 + 2x_4 + \cdots + 4x_{N-1} + x_N) \tag{5.2}$$

Now, we let the function $f(x)$, for example, as $y = x^2$ defined on the interval $[0,\, 10]$ and we will demonstrate this numerical computation of integral. We assume the model with the highest coefficient of determination ($R^2 = 1$), and therefore its values (x_i) approximating function $f(x)$ are equal to the set: $(0^2,\, 1^2,\, 2^2,\, 3^2,\, 4^2,\, 5^2,\, 6^2,\, 7^2,\, 8^2,\, 9^2,\, 10^2)$. The numerical computation of integral from this set of values using Simpson's Rule will be the following:

$$\frac{10-0}{3*10}\left[0^2 + 4(1^2) + 2(2^2) + 4(3^2) + 2(4^2) + 4(5^2) + 2(6^2) + 4(7^2) + 2(8^2) + 4(9^2) + 10^2\right] \tag{5.3}$$

The result of this calculation is equal to $\frac{1000}{3}$, which right approximates the analytical result. We use the method `simps` from the sub-package `scipy.integrate` to compute the approximation of a definite integral of non-parametric models.

5.1.2 Detection Algorithm

The analysis is performed for each non-parametric model and for the best parametric model (models with the highest coefficient of determination R^2) from the given pair of profiles. Algorithm 3 shows the basic principle of running detection methods on a pair of profiles: baseline and target. This algorithm obtains models $\mathcal{M}_b$ and $\mathcal{M}_t$ from profiles $\mathcal{P}_b$ and $\mathcal{P}_t$, and passed it to the `DetectionMethod`, in this case, to the `IntegralComparison` method. Subsequently, this method computes definite integral for both obtained models defined on the interval $[x_{start},\, x_{end}]$. For Equation 5.4 we suppose, that models (both parametric and non-parametric) approximate the function f_b for baseline profile, respectively f_t for target profile. From these computed integrals we calculate the relative error ε that is passed to Algorithm 2:

$$\varepsilon = \frac{\int_{x_{start}}^{x_{end}} f_t dx - \int_{x_{start}}^{x_{end}} f_b dx}{\int_{x_{start}}^{x_{end}} f_b dx} \tag{5.4}$$

Algorithm 2 evaluates this computed value of relative error in comparison to given thresholds. The precise value of thresholds ξ_θ and ξ_Δ has been established based on experiments and requirements of users, specifically for this method. This method returns as the output the members of the structure `DegradationInfo`. A `Result` member contains the individual type of change, that is determined by Algorithm 2 (e.g. NoChange, Optimisation or MaybeDegradation). As an `Error Rate`, it returns the computed value of relative error ε from Equation 5.4. A `Confidence` member contains the coefficient of determination R^2 as a `confidence type` and the minimum value of this coefficient from compared models as a `confidence rate`.

```
Method run_detection_for($\mathcal{P}_b$, $\mathcal{P}_t$)
    foreach $\mathcal{M}_b$, $\mathcal{M}_t \in \mathcal{P}_b$, $\mathcal{P}_t$ do
        $\Delta$ = DetectionMethod($\mathcal{M}_b$, $\mathcal{M}_t$)
        $\vartheta = \theta$
        if $\exists\ \hat{\beta}_0, \hat{\beta}_1 \in \mathcal{M}_b$ then
            $\vartheta = (\mathcal{M}_b \to \vartheta,\ \mathcal{M}_t \to \vartheta)$
        yield DegradationInfo(
            $uid\ = \Delta \to uid,$
            $R^2\ \ = \min \Delta \to R^2_b, \Delta \to R^2_t,$
            $\varepsilon\ = \Delta \to \varepsilon,$
            $\vartheta$
        )
    end
```

Algorithm 3: The general principle of detection performance changes. The input is a pair of profiles (baseline $\mathcal{P}_b$ and target $\mathcal{P}_b$) each containing different kinds of models $\mathcal{M}_b$ and $\mathcal{M}_t$. Subsequently, the specific detection method performs a particular analysis (e.g. integral computation) between two compatible models from each profile (i.e. collected in same way and corresponding to same unique identifier and interval). The result Δ includes the unique identifier *uid*, the kind of parametric model ϑ, the coefficient of determination R^2 and the error rate ε. The coefficients $\hat{\beta}_0$ and $\hat{\beta}_1$ models parametric models in form of $y = \hat{\beta}_1 f(x) + \hat{\beta}_0$.

We will mention that an alternative approach would be to compute the integral $\int_{x_{end}}^{x_{start}} f_t - f_b\,dx$ and analyse the results. This approach will be part of our future work in the frame of improvement the results of this method.

5.2 Local Statistics

When executing the performance detection from given models, it can sometimes be appropriate to divide the data-set into more distinct intervals. We decided to implement another method that adapts this idea because the partial results can provide the user with more precise locations when the changes occur. The second proposed method implements interval-based performance detection. The input of the detection method are two profiles (baseline $\mathcal{P}_b$ and target $\mathcal{P}_t$). The method supports all kinds of models (both parametric and non-parametric).

5.2.1 Initial Algorithm

Again, we perform the analysis on each non-parametric model and the best parametric model for every unique location in both profiles. The input of this analysis are models $\mathcal{M}_b$ and $\mathcal{M}_t$ which are obtained by Algorithm 3 from pair of profiles: baseline $\mathcal{P}_b$ and target $\mathcal{P}_t$. First, the analysis obtains the required set of values from the individual models (values of the model), and the interval on which is this model defined $[x_{start}, x_{end}]$. In the case of parametric models, we compute the values of the model directly from their formulae (respectively its coefficients). The non-parametric models are defined directly by this set of model values. The analysis also checks if the intervals, on which are both models $\mathcal{M}_b$ and

$\mathcal{M}_t$ defined, are of the same length. In the case of an unequal length of compared intervals, we adjust both intervals to the length of the shorter one.

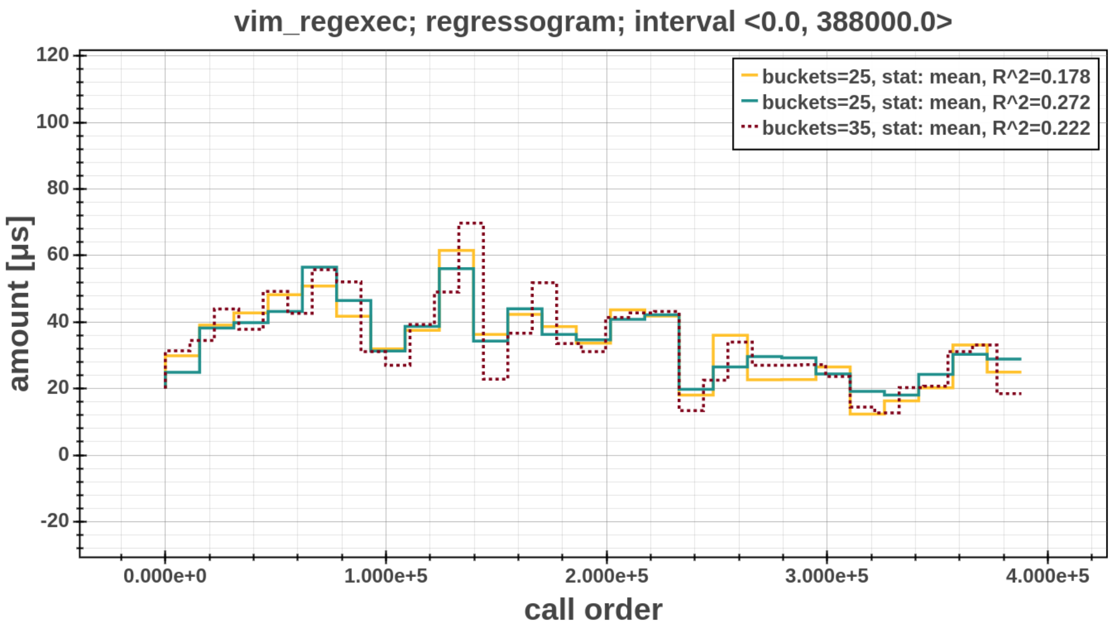


Figure 5.2: The figure shows the demonstration of regressogram unification between two compared profiles. The baseline model is shown in yellow colour, and as displays the legend, it contains 25 buckets. The target model (shown in red dashed model) was initially created with 35 buckets and therefore is not suitable for comparison with the baseline model. A cyan model represents the target model, after the unification with the baseline model, that will be involved in the difference analysis.

This rule of adjustment is applied with an exception of the model created by the *regressogram* post-processor. Since the remaining models approximates each collected value from the interval $[x_{start}, x_{end}]$, they depend only on the length of this interval. On the other hand, the regressogram models depend on the number of buckets, and therefore, the comparison of two models which were created with the different buckets count does not make sense. When regressogram models do not the same count of buckets, we perform their unification, i.e. the target model is adapted to the baseline model (see Figure 5.2). More specifically, that means the original interval of target model $\mathcal{M}_t$ is divided into the new number of buckets according to the number of buckets in baseline model $\mathcal{M}_b$, and thereby we obtain the models with the same number of buckets. We detect the performance changes from these unified buckets, comparing the individual buckets that belong to each other.

5.2.2 Detection Algorithm

A computational logic of this method analyse the individual sub-intervals of the given models: $\underline{b}$aseline $\mathcal{M}_b$ and $\underline{t}$arget $\mathcal{M}_t$, which are defined on the interval $[x_{start}, x_{end}]$. We divide this interval $[x_{start}, x_{end}]$ into k several sub-intervals, where k represents the predetermined number of these sub-intervals. We also have the predetermined minimum number of model values within each sub-interval, with except the last one. Now, we denote the h as the length of each sub-interval, and then we can define the sub-intervals:

$$I_0 = [x_{start}, h], I_1 = [h, 2h], \dots, I_k = [(k-1)h, x_{end}]. \tag{5.5}$$

Let n_j denote the number of model values in interval I_j, and y_i denote the values of the model for $i = x_{start}, \dots, x_{end}$. In other word $n_j = \sum_i I(x_i \in I_j)$ where:

$$I(x_i \in I_j) = \begin{cases} 1, & if\, x_i \in I_j \\ 0, & if\, x_i \notin I_j \end{cases} \tag{5.6}$$

Then, we define Y_j as a statistical metric computed from y_i model values in I_j. For example, we demonstrate the computation of *sum*:

$$Y_j = \sum_{x_i \in I_j} y_i. \tag{5.7}$$

Finally, we define the $m_j = Y_j$ for all sub-intervals I_j:

$$m_j = Y_j I(x \in B_j), \quad for\, j = 0, \dots, k. \tag{5.8}$$

In the same way, we could also defined the remaining metric, that are computed for all sub-intervals I_j: *integral* $\int$, *average* $\overline{X}$, *median* $\tilde{X}$, *maximum* $\mathtt{max}$, *minimum* min, *sum* $\sum$, *first* (P_1) and *second* (P_2) *percentile.* We decided to choose these statistical metrics as appropriate indicators of various changes. At the same time, this set of metrics is easily extensible, since it only needs to implement a statistic method that will accept a 2-D array of values as input.

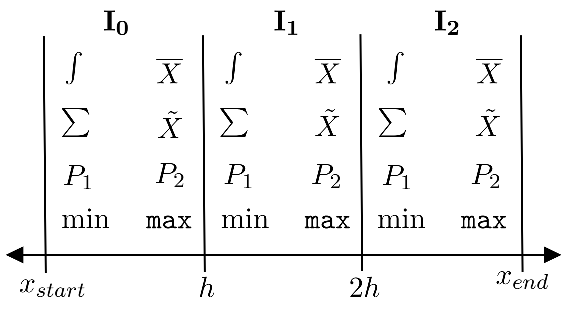


Figure 5.3: The illustration of the computation of the set of statistical metrics within individual sub-intervals (I_0, I_1, I_2).

The final phase follows after the computation of the set of statistical metrics for both baseline and target profiles. We analyse all computed statistical metrics within the individual sub-intervals I_j. First, we compute the relative error against baseline model, which represents the *change indicator* in this method:

$$\varepsilon_j = \frac{m_{t_j} - m_{b_j}}{m_{b_j}} \quad for\; j = 0, \dots, k, \tag{5.9}$$

where m_b contains the results of one computed statistical metrics (e.g. sum) for all sub-intervals I_j from baseline model, respectively m_t from the target model. Then, we use Algorithm 2 to evaluate the values of relative error ε within the individual sub-intervals. We report the performance changes on individual sub-intervals if at least half of the computed statistical metrics are evaluated as changed on relevant sub-interval. Simultaneously, we compute an average relative error for all sub-intervals and report the overall change on the whole interval if its value is higher than the predetermined threshold.

5.2.3 Implementation

This method was implemented using the `NumPy` (Numerical Python) package and its modules. Thanks to an efficient manipulation with `NumPy` arrays and applying mathematical operators on these arrays, we receive a several times faster calculation in comparison to usage of loops and lists [20].

```
at utf_iscomposing:
 0.42x Maybe Degradation (with confidence r_square = 0.72)
   <0.0, 12782.97> 1.69x; <12790.71, 19178.32> -2.24x;
   <25581.42, 31969.03> 2.26x;  <31976.77, 32000.0> -0.75x;
```

Listing 2: The example of the output generated by the `Local Statistics` detection method. It shows the detected change in `utf_iscomposing` function with $0.42x$ Degradation in comparison to the baseline profile. The two last lines in the listings show the member `Partial Integrals`, that shows the detected change in the individual sub-intervals with its error rates.

This method adds a new member to the `Degradation Info` structure, namely `Partial Integrals`. It provides the information about the detected changes at concrete sub-intervals. The user then receives information about the sub-intervals affected by the change, as well as the error rate of this change.

5.2.4 Example

We will show a brief example of the computational logic of this method. Let the count of observations (call order) in collector record be equal to 10. Further, let the baseline and target sets with model values as follows:

- **baseline** model m_b: $[5, 4, 8, 3, 7, 6, 1, 5, 4, 3]$,
- **target** model m_t: $[7, 9, 4, 5, 7, 5, 3, 8, 5, 1]$

These sets of model values with the interval of call order (i.e. x_{pts} equal to $[0, 1, 2, 3, 4, 5, 6, 7,8, 9]$), represent the inputs of the method, that divides this interval and computes the relevant statistical metrics. In this example, we set the minimum number of points within sub-interval to *2* and divide the initial interval into *5* sub-intervals, i.e. let the $i = 4$. Subsequently, we compute the selected metrics for each sub-interval for both models (baseline and target).

In the next phase, we compute the relative error from adjacent values in the Table 5.2 for each computed metric. As shown in Table 5.3, in the interval $[0, 1]$ the relative error for *minimum* Y_i is calculated as:

$$\varepsilon_0 = \frac{Y_{b_0} - Y_{t_0}}{Y_{b_0}} = \frac{7.0 - 4.0}{4.0} \tag{5.10}$$

Algorithm 2 then evaluates the value of relative error and determines the resulting changes. Threshold ξ_θ determines the interval ($<0, \xi_\theta>$) for `No Change` type. Threshold ξ_Δ defines the interval ($<\xi_\theta, \xi_\Delta>$) where the change is detected as uncertain: `Maybe Degradation` or `Maybe Optimisation`. In this example, we set ξ_θ to *0.33* and ξ_Δ to *0.66*. We compute the change score (ΔScore) for each sub-interval from all computed statistical metrics, that determines the resulting changes. The individual types of changes increase or decrease this score, as follows: `Degradation`: $\mathit{\Delta Score} + \mathit{1}$, `Maybe Degradation`: $\mathit{\Delta Score} + \mathit{0.5}$, `Optimisation`: $\mathit{\Delta Score} - \mathit{1}$ and `Maybe Optimisation`: $\mathit{\Delta Score} - \mathit{0.5}$. We use this score to determinate whether the change has occurred or not.

Table 5.1: The table shows the created sub-intervals and the respective values of both models at these sub-intervals.

i =	**0**	**1**	**2**	**3**	**4**
x_{pts}	[0, 1]	[2, 3]	[4, 5]	[6, 7]	[8, 9]
m_b	[5, 4]	[8, 3]	[7, 6]	[1, 5]	[4, 3]
m_t	[7, 9]	[4, 5]	[7, 5]	[3, 8]	[15, 1]

Table 5.2: The table shows the computed metrics on the individual sub-intervals for both compared models. The table rows represent the relevant sub-interval according to the value of x-coordinates according to Table 5.1. The table columns represent the computed metrics within each sub-interval for both compared models (baseline (m_b) and target (m_t)).

	$\int$ / $\bar{\mathbf{X}}$ / $\tilde{\mathbf{X}}$		min		`max`		$\sum$		$\mathbf{P_1}$		$\mathbf{P_2}$	
	b	**t**	**b**	**t**	**b**	**t**	**b**	**t**	**b**	**t**	**b**	**t**
[0, 1]	4.5	8.0	4.0	7.0	5.0	9.0	9.0	16.0	4.3	7.5	14.8	8.5
[2, 3]	5.5	4.5	3.0	4.0	8.0	5.0	11.0	9.0	4.3	4.3	6.3	4.8
[4, 5]	6.5	6.0	6.0	5.0	7.0	7.0	13.0	12.0	6.3	5.5	6.8	6.5
[6, 7]	3.0	5.5	1.0	3.0	5.0	8.0	6.0	11.0	2.0	4.3	4.0	6.8
[8, 9]	3.5	3.0	3.0	1.0	4.0	5.0	7.0	6.0	3.3	2.0	3.8	4.0

Table 5.3: The table shows the computed values of relative error, change score and resulting change within each interval. The colour cells indicate the detected change type within the sub-intervals for the relevant metric.

	$\int$ / $\bar{\mathbf{X}}$ / $\tilde{\mathbf{X}}$	min	`max`	$\sum$	$\mathbf{P_1}$	$\mathbf{P_2}$	**Score**	**Result**
[0, 1]	0.78	0.75	0.80	0.88	0.76	0.78	**8**	**Degradation**
[2, 3]	-0.18	0.34	-0.38	-0.18	0.00	-0.29	**0**	**No Change**
[4, 5]	-0.08	-0.17	0.00	-0.08	-0.12	-0.03	**0**	**No Change**
[6, 7]	0.83	2.00	0.60	0.83	1.13	0.69	**7.5**	**Degradation**
[8, 9]	-0.14	-0.67	0.25	-0.14	-0.38	0.07	**-1.5**	**No Change**

Chapter 6

Experimental Evaluation

In this chapter, we will experimentally evaluate our proposed detection methods using non-parametric models. We will also compare the fitness of new models with models created by existing regression-analysis post-processor. At last, we will compare the run-times of individual post-processors and detection methods over the different test cases.

6.1 Basic Evaluation

We tested novel post-processors on the set of artificial examples consisting of profiles with selected worst-case complexities. For these programs the regression-analysis post-processor will find the following best model: constant, linear, logarithmic, exponential, power and quadratic. These tests aim at verifying the correct functionality of new post-processors compared to an existing post-processor. As shown in Table 6.1, for all tests the implemented post-processors have achieved almost the same accuracy of estimate as to the regression-analysis post-processor. The processing time in these examples was negligible (in the range from *0.02s* to *0.10s*). In these examples we used the regressogram model with 10 buckets, moving average model with window width equal to 300, and kernel regression model that used the Epanechnikov kernel (see Figure 3.1a) and Scott's rule (see Section 3.4.1) to determine optimal kernel bandwidth.

Table 6.1: The comparison of fitness of models created by all available PERUN's post-processors on the set of artificial examples. The table rows represent individual examples that have the stated algorithmic complexity (e.g. constant $\mathcal{O}(c)$, logarithmic $\mathcal{O}(n \log n)$, etc.) and columns represents the fitness of best models of individual post-processor. The fitness of models is expressed by the value of the coefficient of determination R^2.

	reg-analysis	**regressogram**	**moving-avg**	**kernel-reg**
constant	1.000	1.000	1.000	1.000
linear	0.999	0.989	0.993	0.996
logarithmic	0.999	0.821	0.802	0.843
exponential	0.999	0.981	0.977	0.987
power	0.972	0.989	0.991	0.995
quadratic	0.999	0.987	0.989	0.994

In the next experiment, we compared the accuracy of estimates created by post-processors and their processing time on more complex profiles, that we collected on VIM binary with different collecting configurations. We evaluated our solutions on two profiles, which includes *25* different collected functions. The first profile (#1) contains records with the number of samples in the range from 3 to 3 290 469 samples (773 711 on average) and second profile (#2) in the range from 2 to 186 886 samples (42 038.55 on average). As shown in Table 6.2, the new non-parametric post-processors achieved better fitness in comparison to models created by original regression analysis post-processor. Kernel-regression post-processor had the slowest post-processing times, however, it achieved the best fitness of the resulting models. For this example, we used the following configurations of post-processors: the regressogram model with Doane's method to determine optimal bucket counts; the simple moving average model with window width equal to 4; and kernel regression model with kernel-ridge mode, where the γ parameter was set to $\frac{1}{n}$, where n represents the count of the samples.

Table 6.2: The comparison of post-processing times $t[s]$ and coefficient of determination R^2 between the available post-processors within PERUN. The coefficient of determination was computed as the average value of these models, which were present in the tested profiles. Highlighted cells represent the extremes in both compared metrics.

	reg-analysis		**regressogram**		**moving-avg**		**kernel-reg**	
	R^2	**t**[s]	R^2	**t**[s]	R^2	**t**[s]	R^2	**t**[s]
#1	0.192	122.90	0.117	45.60	0.551	71.78	0.786	126.68
#2	0.222	11.01	0.415	3.86	0.572	5.28	0.775	76.71

6.2 Degradation Detection on Vim

We evaluated our detection methods on repository VIM, which contains known reported performance issue. We tried to detect known issue, that was reported in version `v7.4.2293` and resolved in `v8.0.0190`[1]. In addition to the detection of this issue, we compared two following version and checked our assumption, that they had no significant changes. This known issue caused the performance degradation of specific functions because it used the type `garray_t` instead of `hashtab_t` to collect tags. Since the tags are stored in a `garray_t` structure, VIM has to perform a linear search of all existing tags every time a new tag is added. Using a hash table (structure `hashtab_t`) drastically improves the speed and resolves this performance issue.

Collection Strategy. We used a `trace` collector as the described issue affects only specific functions. This collector allows one to measure run-times of functions that are executed during the run and subsequently to detect the changes between them. As the first step, we have to collect (generate) the tags (taglist) using `Ctags`[2] utility (e.g. ctags -R /usr/include/). For the reproduction of the described issue we need to run VIM with the following configuration:

[1] https://github.com/vim/vim/commit/810f9c361c83afb36b9f1cdadca2b93f1201d039
[2] http://ctags.sourceforge.net/

```
#!/bin/bash
perun collect --cmd="./src/vim" --args="-u NONE"
    --workload="--cmd 'echo len(taglist("a"))' --cmd q" trace
```

Listing 3: The complete PERUN's command to collect the profiles by `trace` collector with a specifically described strategy for VIM binary `./src/vim`: an argument `-u` with value `NONE` ignores vim configure files (`.vimrc`); the command `echo len(taglist("a"))`, which finds all tags containing the letter `"a"`; and at the end the command `:q` to terminate the VIM editor.

6.2.1 Initial Experiments

Collecting Phase. We first ran this configuration without further specifications of traced functions, i.e. we collected run-times of each called function. But we found out that running the VIM with this configuration is complex and calls a `377` traced functions resulting into a large number of collect records. The generation of this profile lasted more than `2 hours`, and the resulting profile had the size of **214 MiB**. Such extensive profiles are difficult to process, and therefore we needed some level of abstraction if we wanted to collect reasonable amount of data. Therefore, we used the `trace` collector with the sampling, where we set the global sampling of calling each function, so we would monitor every n^{th} calling only. For this example, we set the value of `global-sampling` to `500` (tracing each 500^{th} call), and the resulting profile was collected in `419.75 seconds` with the size of `22 MiB` for the version `v7.4.2293`, and `2.3 MiB` for `v8.0.0190`.

Evaluation. From these collected profiles we subsequently create a performance models, by our post-processors, and then we used detection methods to determine possible changes. We tested each type of non-parametric models — regressogram (**RG**), moving average (**MA**) and kernel regression (**KR**) — and two new detection methods: Integral Comparison (**IC**) and Local Statistics (**LC**). Recall that we want to detect known issue between versions `v7.4.2293` and `v8.0.0190` and compare two following versions (`v8.0.0190` and `v8.1.0000)` that there is no significant performance change. Therefore we chose the version `v8.0.0190` as the baseline profile and remaining versions as target profiles in these comparisons. In the first experiment (`v8.0.0190 - v7.4.2293`) we compared **333** common functions from both profiles and in the second case (`v8.0.0190 - v8.1.0000)` we compared in summary **388** functions.

As shown in Table 6.3, in the second case we detected only minimal count of functions in which change has occurred. Most of the changes were reported using the *regressogram* model, which confirms our assertion that it is an over-estimating method in some situations. The results of the comparison between versions `v8.0.0190 - v8.1.0000` show that the difference between these two versions should be stable. On the contrary, the first comparison (that contains the version where we expected performance degradation) includes on average **7%** more functions with the detected change. This average increase of detected changes confirms the suspicion of performance issue because its value approximately corresponds to the impact of this issue on specific measured functions.

Table 6.3: The results of our first experiments of detecting degradations in Vim. Each cell shows how many functions were reported as degradations (-) or optimisations (+). We use (?) to signify that some change was detected, but was not drastic enough to be reported. Colour of cells reflects our expectations: green denotes meeting our expectations and red its failing.

		v8.0 - v7.4			**v8.0. - v8.1**		
	Model	-	+	?	-	+	?
IC	**RG**	18% (60)	17% (56)	13% (43)	11% (36)	8% (26)	13% (43)
	MA	5% (16)	3% (10)	0% (2)	0% (2)	1% (4)	0% (0)
	KR	6% (20)	2% (6)	0% (1)	0% (3)	1% (4)	0% (1)
LC	**RG**	11% (36)	8% (26)	13% (43)	6% (20)	3% (10)	6% (21)
	MA	13% (43)	8% (27)	11% (35)	6% (20)	3% (10)	6% (21)
	KR	12% (40)	6% (21)	11% (36)	6% (20)	3% (12)	7% (24)

6.2.2 Further Experiments

Collect Phase. However, the initial experiments were not focused on the specific functions, and therefore we decide to continue in the detection of the mentioned issue in Vim. In the following experiments, we aimed to focus on functions, that are directly involved in the issue. We analysed this issue and we located the function that performs the main logic of the command which we are executing (echo len(taglist("a"))). This function is located in the module `tag.c` and is called `find_tags`[3]. We collected the set of functions, which we will analyse, from the call-graph of this main function. We obtained this call-graph using a `cflow`[4] tool. Subsequently, we made an intersection of this set of functions with the functions, which were present in the profile with actually collected records. We further selected out these functions, which were called at least three times during the whole run. The resulting set of functions contains the **25** specific functions (without specification was their count **333** or **388**). The new collecting data by `trace` collector for this set lasted **475.48 seconds**, and the collected profile had a size **217.57 MiB**. The size of this profile is almost equal to the first collected profile (**214 MiB**), what it is due to the fact that in this case, we did not use sampling. When we tried using the specific sampling according to the number of calls of each function, then the resulting profile was generated in **87.34 seconds** and had a size **13.23 MiB**. However, the subsequent detection of the performance changes was distorted by the number of samples, and the processing time did not increase significantly compared to the profile without sampling (since we analysed **25** functions).

Evaluation. We post-processed collected profiles (without and with samples) with all available post-processors. We tested each type of non-parametric models (as in Section 6.2.1) and also models created by regression-analysis post-processor (**RA**). Subsequently, we use the available methods to detection potential performance changes from created models. In this evaluation we present detection methods presented in this thesis: Integral Comparison (**IC**) and Local Statistics (**LC**); and Average Amount Threshold (**AAT**).

Since the results of analysis by individual detection methods do not have unified format, and the structure `Degradation Info` includes different members, these results cannot directly compared in these experiments. Therefore, we manually analysed the results of all

[3]https://github.com/vim/vim/blob/master/src/tag.c#L1546
[4]https://www.gnu.org/software/cflow/

detection methods which compared the profiles collected from the VIM versions `v8.0.0190` and `v7.4.2293`. Most of the detection methods detected the performance changes at functions listed in Table 6.4. In the remaining **22** functions they detected no changes, or eventually in some cases insignificant changes with a small value of error rate.

Table 6.4: The results of comparison of collected profiles on the specific set of the functions of VIM. We show the detected error rates in the individual functions using implemented detection methods IC and LC. These methods performed detection on all post-processing models (RG, MA, KR and RA). Colour of cells reflects the extremes of confidence rate, in this experiment R^2 showed in parenthesis, between the models. The last column shows the results of comparison by Average Amount Threshold (AAT) method, that works with raw performance data, i.e. with the collected run-times of these functions.

		RG	**MA**	**KR**	**RA**	**Raw**
ga_grow	**IC**	1.00× (0.10)	1.00× (0.52)	1.00× (0.92)	1.00× (1.00)	0.88×
	LC	0.23× (0.08)	1.18× (0.55)	0.55× (0.78)	0.86× (1.00)	
ga_init2	**IC**	1.00× (0.08)	0.71× (0.46)	0.70× (0.96)	0.66× (0.15)	0.47×
	LC	0.16× (0.06)	0.11× (0.62)	0.26× (0.28)	0.60× (0.12)	
ga_clear	**IC**	0.26× (0.15)	0.46× (0.45)	0.13× (0.88)	0.61× (0.07)	0.34×
	LC	0.28× (0.10)	0.27× (0.45)	0.54× (0.93)	0.12× (1.00)	

In the Table 6.4, the values of error rates are represented by the relative error that is computed in a specific way by the individual detection methods. It indicates the rate of the change of target model in comparison to the baseline model. For example, an error rate equal to 1× can represent the degradation from $100ms$ in the baseline profile to $200ms$ in the target, however, this would only apply in the case of run-times comparison. Our detection methods (IC and LC) works with the models, from which compute the individual change indicators (such as integral or other statistical metrics), i.e. the error rate can represent the change of areas under models (integral) and so on.

As shown in Table 6.4, all functions name starting with prefix `ga_` are working with the structure `garray_t`. This structure caused performance issue in vim `v7.4.2293` until its replacement with structure `hastab_t` to collect tags in version `v8.0.0190`. The average error rates in these functions and their confidence detection rates, expressed by the coefficient of determination R^2, which were detected from the non-parametric models are the following: `ga_grow`: **0.85**×, (0.62), `ga_init2`: **0.53**×, (0.34) and `ga_clear`: **0.34**×, (0.52). Table 6.4 shows that these results derived from the non-parametric models are almost the same to results, that were created from the raw performance data by AAT method, i.e. outour results were not affected by the fitness of the models or inaccuracy of the detection methods. We assume that detected results are related to mentioned issue because after the replacement of problem structure has improved the performance of these functions in VIM.

6.3 Detection Method Evaluation

In this thesis, we presented two new detection methods, namely: Integral Comparison and Local Statistics. We compared these methods with the existing methods, which are already implemented within PERUN. Most of these methods were described in detail in [17]. These methods, however, work only with regression models or raw resource data, and therefore we will not compare their functionality but we will compare only processing time on the different data-sets. The first profile (**#1**) includes **25** different functions, where the average count of samples in one function is equal to **768 358**. The second profile (**#2**) includes a greater number of functions, but with the fewer samples per each function (**3 188** on average).

Table 6.5: The comparison of processing times (in **seconds**) of individual detection methods on different collected profiles. We compare all implemented methods within the Perun framework: Average Amount Threshold (**AAT**), Best Model Order Equality (**BMOE**), Linear Regression (**LREG**), Polynomial Regression (**PREG**), Fast-Check (**FCH**) and two new implemented detection methods Integral Comparison (**INT**) and Local Statistics (**LOC**).

	AAT	**BMOE**	**LREG**	**PREG**	**FCH**	**INT**	**LOC**
#1	512.63s	0.10s	77.47s	0.12s	75.05s	0.62s	1.80s
#2	10.94s	0.05s	7.86s	0.08s	6.98s	0.15s	0.68s

As shown in Table 6.5, the processing times of our new methods (**INT** and **LOC**) are stable and both achieve good times under any conditions, while other methods (**AAT**, **LREG** or **FCH**) have fluctuating processing times depending on the number of compared functions or samples. This is due to the fact, that these methods work with a full profile, while our new methods work only with regression models.

The Average Amount Threshold (**AAT**) method groups all of the collected resources to the unique identifier (e.g. function name) and then computes the averages of resource amounts. The Best Model Order Equality (**BMOE**) chooses the best parametric model (i.e. the one with highest coefficient of determination R^2) from both compared profiles (baseline and target) and compare it lexicographically (e.g. the linear model is lexicographically smaller than quadratic model). The Fast Check (**FCH**) method is based on the subtraction of best parametric models and subsequently interleaving these data by newer models of regression-analysis post-processor. The Linear Regression (**LREG**) method analyse the coefficients of linear regression models, respectively the coefficients of their subtraction. The last method, namely Polynomial Regression (**PREG**), uses polynomial regression to quantify the rate of the change, i.e. it represents the change in the form of n^{th} degree polynomial function.

Final Evaluation. Our experiments show that implemented post-processors can interleave the data more adequately and faster in comparison to existing post-processor of regression analysis in many cases. Implemented detection methods detected the performance degradation between two versions of Vim at three specific functions, which works with the structure that caused a known issue. These methods were successful in comparing the detection methods, where they achieved one of the best processing times, without any dependency on the input data-set or specific models.

Chapter 7

Conclusion

In the thesis, we have presented the post-processors for performance data modelling which are based on the non-parametric techniques: regressogram, moving-average and kernel regression. Further, we have proposed two methods for automatic detection of performance changes, mainly focused on the non-parametric models. We have implemented these post-processors and detection methods as the components of the PERUN framework. Our post-processors achieved three times faster post-processing time and almost triple improvement in the fitness of the models in comparison to the existing post-processor within the PERUN. Moreover, we have evaluated our solutions on the different versions of VIM, which contains a known performance issue. We have detected the performance degradation of three specific functions, which are related to this issue. The results of this thesis were also presented at students conference Excel@FIT'19, where we got the award from the general sponsor of this conference for extraordinary work with great benefits to the practice.

Our future work will focus on increasing the accuracy and efficiency of our approach for the automatic detection of performance changes. We will focus on improving the performance data collector, as well as the new types of collectors with an option to measure several metrics and support for different programming languages (such as Java). Further, we will want to improve the performance of post-processors for faster processing of collected profiles, and mainly we will want to bring new solutions to post-processing performance data. New solutions should extend the range of input data distribution that can be adequately approximated to achieve more accurate performance detection. Also, we want to focus on the precision of pinpointing the actual problem source to prepare the solution that could be effectively used in a broad range of projects. Finally, we plan to evaluate our solution on the existing projects (we target on RUBY) and potentially detect real performance bugs.

Bibliography

[1] Chen, Y.-C.: Introduction to Nonparametric Statistics: Regression: Regressogram and Kernel Regression. Technical report. University of Washington. 2018.
Retrieved from:
http://faculty.washington.edu/yenchic/18W_425/Lec9_Reg01.pdf

[2] Fajmon, B.; Hlavičková, I.; Novák, M.; et al.: Numerická matematika a pravděpodobnost (Informační technologie). 2014. [Online; accessed 13.5.2019].
Retrieved from: http://matika.umat.feec.vutbr.cz/inovace/texty/INM/CZ/INM_plna_verze_CZ.pdf

[3] Hurvich, C. M.; Simonoff, J. S.; Tsai, C.-L.: Smoothing Parameter Selection in Nonparametric Regression Using an Improved Akaike Information Criterion. *Journal of the Royal Statistical Society. Series B (Statistical Methodology)*. vol. 60, no. 2. 1998: pp. 271–293. ISSN 13697412, 14679868.
Retrieved from: http://www.jstor.org/stable/2985940

[4] Jones, E.; Oliphant, T.; Peterson, P.; et al.: SciPy: Open source scientific tools for Python. 2001–. [Online; accessed 29.4.2019].
Retrieved from: http://www.scipy.org/

[5] Klemelä, J.: *Multivariate Nonparametric Regression and Visualization: With R and Applications to Finance*. Wiley Series in Computational Statistics. Wiley. 2014. ISBN 9781118593509.
Retrieved from: https://books.google.cz/books?id=8gWIAwAAQBAJ

[6] Koláček, J.: *Jádrové odhady regresní funkce [online]*. Disertační práce. Masarykova univerzita, Přírodovědecká fakulta, Brno. 2005 [cit. 2019-04-05].
Retrieved from: https://is.muni.cz/th/no1rh/

[7] McKinney, W.: Pandas: Powerful Python Data Analysis Toolkit. 2019. [Online; accessed 13.5.2019].
Retrieved from: http://pandas.pydata.org/pandas-docs/stable/pandas.pdf

[8] Pavela, J.: *Knihovna pro profilování datových struktur programů C/C++*. Master's Thesis. BUT, FIT. 2017.
Retrieved from: http://www.fit.vutbr.cz/study/DP/BP.php?id=20045

[9] Pavela, J.; Stupinský, Š.: Towards the detection of performance degradation. In *Excel@FIT'18*. Brno, Czech Republic. 2018.
Retrieved from: http://excel.fit.vutbr.cz/submissions/2018/016/16.pdf

[10] Pedregosa, F.; Varoquaux, G.; Gramfort, A.; et al.: Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research*. vol. 12. 2011: pp. 2825–2830.

[11] Perun: Performance Version System. [Online; visited 15.3.2019].
Retrieved from: `https://github.com/tfiedor/perun`

[12] Racine, J.: *Nonparametric Econometrics: A Primer*. Foundations and trends in econometrics. Now Publishers. 2008. ISBN 9781601981103.
Retrieved from: `https://books.google.cz/books?id=UgFcW9VVjRMC`

[13] Barbier de Reuille, P.: PyQt-Fit: Regression toolbox in Python. 2013. [Online; accessed 29.4.2019].
Retrieved from: `https://pythonhosted.org/PyQt-Fit`

[14] Scott, D. W.: On optimal and data-based histograms. *Biometrika*. vol. 66, no. 3. 1979: pp. 605–610.
Retrieved from: `https://academic.oup.com/biomet/article/66/3/605/232642`

[15] Seabold, S.; Perktold, J.: Statsmodels: Econometric and statistical modeling with python. In *9th Python in Science Conference*. 2010.

[16] Shalizi, C.: Nonparametric Regression. Technical report. Carnegie Mellon University. 2015.
Retrieved from:
`http://www.stat.cmu.edu/~larry/=stat401/NonparametricRegression.pdf`

[17] Stupinský, Š.: *Automatic Detection of Performance Degradation*. Project practice. Brno University of Technology, Faculty of Information Technology. 2018.

[18] Stupinský, Š.: Non-Parametric Modelling for Automatic Detection of Performance Changes. In *Excel@FIT'19*. Brno, Czech Republic. 2019.
Retrieved from: `http://excel.fit.vutbr.cz/submissions/2019/052/52.pdf`

[19] Sturges, H. A.: The choice of a class interval. *Journal of the american statistical association*. vol. 21, no. 153. 1926: pp. 65–66.
Retrieved from: `http://www.esalq.usp.br/departamentos/lce/arquivos/aulas/2013/LCE0216/Sturges1926.pdf`

[20] van der Walt, S.; Colbert, S. C.; Varoquaux, G.: The NumPy Array: A Structure for Efficient Numerical Computation. *Computing in Science Engineering*. vol. 13, no. 2. March 2011: pp. 22–30. ISSN 1521-9615. doi:10.1109/MCSE.2011.37.

[21] Wand, M.; Jones, M.: *Kernel Smoothing*. Chapman & Hall/CRC Monographs on Statistics & Applied Probability. Taylor & Francis. 1994. ISBN 9780412552700.
Retrieved from: `https://books.google.cz/books?id=GTOOi5yE008C`

[22] Zakamulin, V.: *Market Timing with Moving Averages: The Anatomy and Performance of Trading Rules*. New Developments in Quantitative Trading and Investment. Springer International Publishing. 2017. ISBN 9783319609706.
Retrieved from: `https://books.google.cz/books?id=uSU_DwAAQBAJ`

Appendix A
Storage Medium

`/perun/*` — source code of Perun from date May 15, 2019

`/README.txt` — useful information about the storage medium content

`/text/*` — source code of this thesis

`/xstupi00.pdf` — final version of this thesis