BRNO UNIVERSITY OF TECHNOLOGY FACULTY OF INFORMATION TECHNOLOGY

Excel@FIT 2019
http://excel.fit.vutbr.cz

Non-Parametric Modelling for Automatic Detection of Performance Changes

Šimon Stupinský*

Abstract
Current tools that manage project performance do not provide a satisfying evaluation of the overall performance history, which is often crucial when developing large applications. In our previous work, we introduced a tool-chain that collected set of performance data, extrapolated these data into a performance model represented as a function of two depending variables, and compared the result with the model of the previous version reporting possible performance changes. The solution was, however, dependent on precisely specifying and measuring those dependent variables. In this work, we propose more flexible approach of computing performance models based on collected data and subsequent check for performance changes that requires only one measured kind of variable. We evaluated our solution on different versions of `vim`, and we were able to detect a known issue in one of the versions as well as verify that between two stable versions there were no significant performance changes.

Keywords: Performance — Continuous integration — Non-parametric analysis — Regressogram — Moving average — Kernel regression — Automated changes detection — Difference analysis

Supplementary Material: Project Repository

*xstupi00@fit.vutbr.cz, *Faculty of Information Technology, Brno University of Technology*

1. Introduction

Performance testing is a critical factor in optimization of programs. This form of non-functional testing aims to determine the performance of a system under certain conditions to identify its critical locations, but is still not so well developed. In contrast, *functional* testing is covered by many tools as a part of a continuous integration. Although there exists several high-quality tools for performance testing, the fully automating of the profiling resources, or the subsequent comprehensive management of the created performance, is provided only by some of them.

Maintaining optimal performance during development requires tracking multiple, often conflicting, aspects under changing conditions. A user has to start performance regression tests, handle the testing history and the context of the executed performance tests himself. However, manual manipulation with a large amount of data is highly prone to error and may lead to loss of the exact history of tracked changes.

In particular, existing systems do not offer the automation and are missing more precise context for performance management. When managing performance profiles without full automation, the user is forced to annotate and manage all collected data manually. Possible solutions for some of these problems could be removed by using a database to store collected information and performance profiles, but by using it, we have to face new complications. Hence, we would like a system, which will be light-weight, distributed and be able to manage collected data and performance profiles.

To manage performance version the research group *VeriFIT* developed the tool *Perun*: *Performance under control* [1]. It provides full functionality for automation of the profiling process, manages and stores collected data, and allows effective processing of the results, as well as a set of visualisation techniques. Nowadays, our framework for automated detection of performance changes involves different types of collectors, post-processor yielding models obtained by the regression analysis and several detection methods. This way we can provide estimates of performance changes, such as degradation or optimisation, during the code development for a broad range of programs.

Moreover, since *Perun* works as a wrapper over repository, we obtain a powerful difference analysis. Whenever when the user releases a new version of the project, Perun runs the following. First, the profiles for the new version are generated, and the *regression models* are created by post-processors. Subsequently, Perun compares the newly regression models with stored best models of the previous stable version. Finally, the user obtains the list of performance changes, where each change includes three crucial aspects: *precise location*, *severity* and *confidence* [2].

The post-processing is one of the most important aspects to achieve the most accurate performance analysis of collected data. The models created by individual post-processors are the basis for our detection methods. But, the current post-processor that implements *regression-analysis* requires to find the right so called independent variable. This regression analysis is based on the prediction of the *dependent* variable values for every value of the *independent* variable. However, this assumption, that an unknown function belongs to the class of functions dependent on the parameter is sometimes not fulfilled and therefore the resulting detection by these models can also be affected. Sometimes the data are simply not dependent on anything, and we still need to have an excellent model to achieve more precise results. Hence, we propose to implement new types of post-processors, that will be based on the *non-parametric* methods.

New non-parametric modelling brings new possibilities of post-processing data and subsequent detection of performance changes by using them. In particular, in this work we propose the implementation of three new post-processors: *regressogram*, *moving average* methods and *kernel regression* and two detection methods: *integral comparison* and *local statistics*, within the Perun framework. Models created by these post-processors and new detection methods allow performing the detection of changes more flexibly and thereby could possible achieve better results in some cases in the whole process of automatic detection.

2. Post-processing the data

The raw performance data obtained by collectors have no significant value without further processing. Therefore we post-process data so they are suitable for difference analysis. We use the statistical technique called smoothing [3] which can be approached in two ways — *parametric* or *non-parametric*. Parametric estimates are based on the assumption that the unknown function belongs to the class of functions that depends on some parameters. Non-parametric estimates do not prescribe the data of ”Procrustes lies” of parametrisation, but let the data speak themselves.

Specifically in the context of our framework, we use estimates of the *regression function*. The goal of *regression analysis* is to find appropriate approximation $\hat{f}$ of an unknown function f. The statistical problem, which is solved by this analysis, is the fitting of the curves to currently processed data-set of points. The purely parametric approach does not always satisfy the need for flexibility, but still, it is useful and retains its benefits. The example of the parametric estimate of regression function can be the regression curve reflecting the linear dependency. In spite of data processing development, both approaches preserve their advantages and are orthogonal to each other.

In the next sections, we will use the following notation of X variable and Y variable. In our framework, the X variable can be represented as the *size of the underlying structures* and Y variable as *function run-time*.

2.1 Regression Analysis Post-Processor

This post-processor (authored by Jiří Pavela) implements the parametric methods, which are used to determine the relationship between the *dependent* (Y) and *independent* variables (X). *Regression function*, as a result of this analysis, then expresses the relationship between these two variables. In general, the result of this analysis is the set of mathematical functions that describe the behaviour of code functions in the researched program (i.e. consumed memory to the size of a data structure).

$$\hat{\beta}_0 = \frac{\sum y_i - \hat{\beta}_1 \sum f(x_i)}{n} \tag{1}$$

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

We use the *coefficient of determination* (R^2) to rate the goodness-of-fit of *regression functions*. Currently, we supported selected types of regression models in this post-processor: *constant*, *logarithmic*, *quadratic*, *power*, etc. All models, except quadratic, are computed using the general Formulae 1, for coefficients $\hat{\beta}_0, \hat{\beta}_1$ of function model $f(z) = \hat{\beta}_0 + \hat{\beta}_1 z$. A quadratic model uses specific computation Formulae 2, since it requires more coefficients. The implementation of this post-processor is already covered to a great extent in our previous works [4, 2].

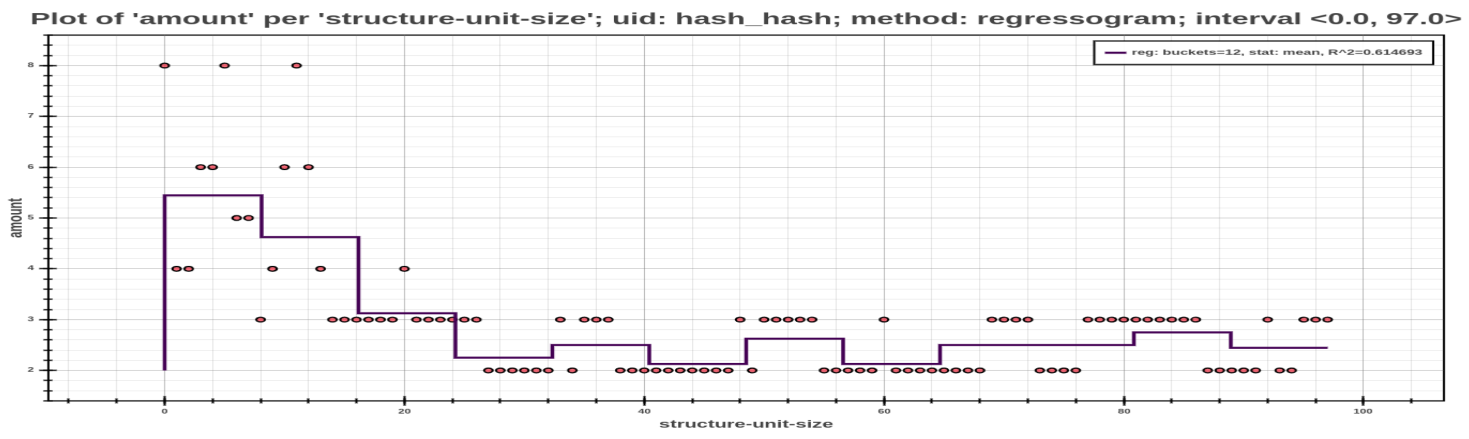

Figure 1. Example of regressogram model created with default value of options.

3. Regressogram Post-Processor

The first proposed post-processor, implements the simplest non-parametric method called *regressogram*. This method, also called constant function by steps or binning-approach, uses the same idea as a histogram at density estimation. Similar to the histogram, we first choose N that represents the number of buckets. The main idea is to divide the set of values X variable into the N intervals $B_n, for j = 1, 2, \dots, N$ with an equal width:

$$B_1 = \left[0, \frac{1}{N}\right), B_2 = \left[\frac{1}{N}, \frac{2}{N}\right), \dots, B_N = \left[\frac{N-1}{N}, x_{max}\right) \tag{3}$$

Subsequently the estimate in the point $x \in B_j$ is taken as the mean of values Y on this sub-interval, where $I_{|B_j|}$ signs the indicator function of sub-interval B_j:

$$\hat{f}(x,h) = \frac{\sum_{i=1}^{n} Y_i I_{|B_j|}(x_i)}{\sum_{i=1}^{n} I_{|B_j|}(x_i)}, \tag{4}$$

The fitness of estimation of regressogram model depends primarily on the number of buckets into which the interval of x-coordinates divided. Our post-processor provides two options. The user can either choose a number of buckets manually or use one of the supporting methods to estimate the optimal number of buckets. These methods are well founded in the documentation of the `SciPy` package [1]. These are simply plug-in methods that give good starting points for number of regressogram buckets. Post-processor also provides the option to choose the statistics function to compute the resulting value within buckets, with *mean* and *median* as two primary statistics metric. Generally, the resulting estimate by regressogram model appropriately describes the functions shape, but the estimate can be too thick for further processing.

The implementation of this post-processor is based on `SciPy` statistics package, which contains the method for executing a *binning approach*. All the necessary integration within our framework with friendly usage with many other options.

[1]https://docs.scipy.org/doc/numpy/histogram/bucket_edges

4. Moving-Average Post-Processor

The natural generalisation of regressogram is the *moving average*, so-called *rolling mean* or *running average*. This method uses local averages of Y values, with the estimate at X based on the centering around this point. It is based on the assumption, that if f is an unknown smooth function, then observed points X_i near point X contains information about the value f function at this point. Among the useful properties of moving average belongs the ability to reduce the effect of temporary variations in data, or an ability to show the data trend more clearly. These methods can detect the outliers in estimated data-set, as it can highlight any value above or below the trend. We decided to implement two most popular and most used variants: *Simple Moving Average* and *Exponential Moving Average*.

A *Simple Moving Average (SMA)* is calculated by average Y values over the specified period of X interval. It is an unweighted moving average method, i.e. each *x-coordinate* in the estimated data-set has equal importance and is weighted equally. In science and engineering, the average is usually calculated from an equal number of data on either side of a central value and the same principle was implemented in our processor. Besides, our processor offers option to calculate the average from the previous N samples, where the N marks the width of the period in both cases. Despite, it is an unweighted moving average, our processor supports different types of *window*, that has the role of *weighted function*. Their more specific definition and the visual view is available in `SciPy` documentation [2]. The last extension that is provided in this method is associated with boundaries of interval. Since the window at boundaries does not contain enough points usually, users have the option to specify the minimal count of points in the period to calculate correct results.

An *Exponential Moving Average (EMA)* is a weighted moving average that places a higher weight on

[2]https://docs.scipy.org/doc/scipy/window_types

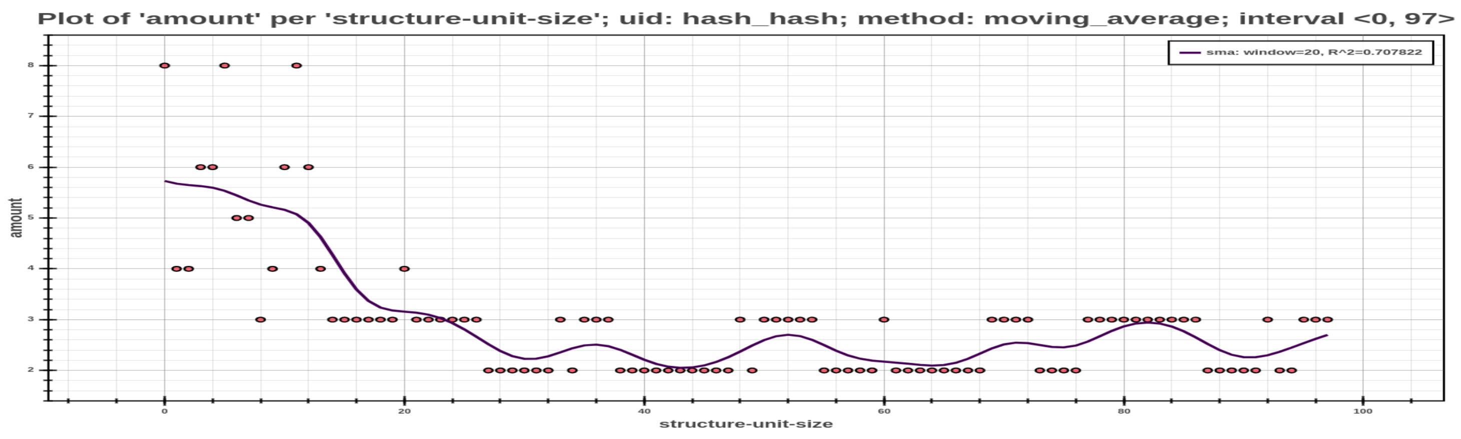

Figure 2. Example of moving average model created by this post-processor with custom value of options.

the most recent data points. Compared with *SMA* it responds more quickly to recent changes in the current interval. The x-coordinates have assigned the weight, that decreases exponentially, never reaching zero. Its formula involves using a multiplier and starting with calculating an *SMA* over a particular sub-interval. Subsequently, the multiplier for weighting the *EMA* must be calculated depending on the selected formula, which returns specific value of coefficient α. The different ways of computation this coefficient are available by supported options for a specific parameter that is derived from the so called `decay` parameter [3]. This coefficient represents the degree of weighting decrease, a constant smoothing factor, when higher value discounts earlier observations faster and smaller value to the contrary.

5. Kernel Regression Post-Processor

A *Kernel Regression* is a *non-parametric* technique that estimates the conditional expectation of a random variable [5]. The objective is to find a *non-parametric* relation between a pair of random variables X and Y. All types of kernel estimates depend on the type of *kernel*, that has the role of *weighted function* and on the *width* of the smoothing window, that controls the smoothness of the estimate. It puts the kernel to each observation point of data-set and then assigns the weight to each point depending on the distance from the currently estimated data point. Generally, the kernel estimates of the regression function f at point x can be defined as is shown on the Formula 5, where function

$$\hat{f}(x,h) = \sum_{i=1}^{n} W_i(x,h)Y_i, \tag{5}$$

$W_i, for\ i = 1,2,\ldots,n$, are called weights and are independents on Y, but depends on positive number h, that is called smoothing parameter. Specifically type W depends on kernel function K. Among the most popular types of the kernel estimates we can includes *Nadaraya-Watson* estimates, which are used by our post-processor. The whole set of kernel estimates of regression function (*Nadaraya-Watson*, *Priestley-Chao*, etc.) are asymptotically equivalent [5]. In most of post-processor modes we use *Nadaraya-Watson* estimates, and therefore we defined these estimates as a weighted function of shape:

$$\hat{f}_{NW}(x,h) = \frac{\sum_{i=1}^{n} K_h(x-x_i)Y_i}{\sum_{i=1}^{n} K_h(x-x_i)}. \tag{6}$$

The selection of concrete kernel is not essential from an asymptotic point of view [6]. It is advisable to choose the optimal kernel because these kernels are continuous on $\mathbb{R}$ and the estimated regression function then inherit the kernel smoothness. This post-processor supports a few types of kernels, concretely *Gaussian*, *Tricube*, *Epanechnikov* and two kernels of the higher order, *Gaussian* and *Epanechnikov* of order 4. On the Graph 4 we can see sketches of individual supported kernels. A *Gaussian* kernel is less steep, and hence the resulting kernel model will reflect the greater count of surrounding points with decreased weight from its centre. On the other hand, the *Tricube* and *Epanechnikov* kernels put more emphasis on the currently estimated point and reflect less count of surrounding points with bigger weights.

We provide multiple modes, which does not differ in the resulting estimate. Individual modes are different in the way of computation of the resulting kernel estimate. The purpose of these modes is to provide flexibility in constructing the kernel estimate. Moreover, user has the option to choose from available kernel smoothing methods, as well as various combinations of the estimate. One of the modes provides three methods: local-polynomial regression, local-linear regression and spatial-average method. Information about mentioned methods is available in documentation [4].

[3]https://pandas.pydata.org/pandas-docs/ewma

[4]https://pythonhosted.org/PyQt-Fit/kernel_regression

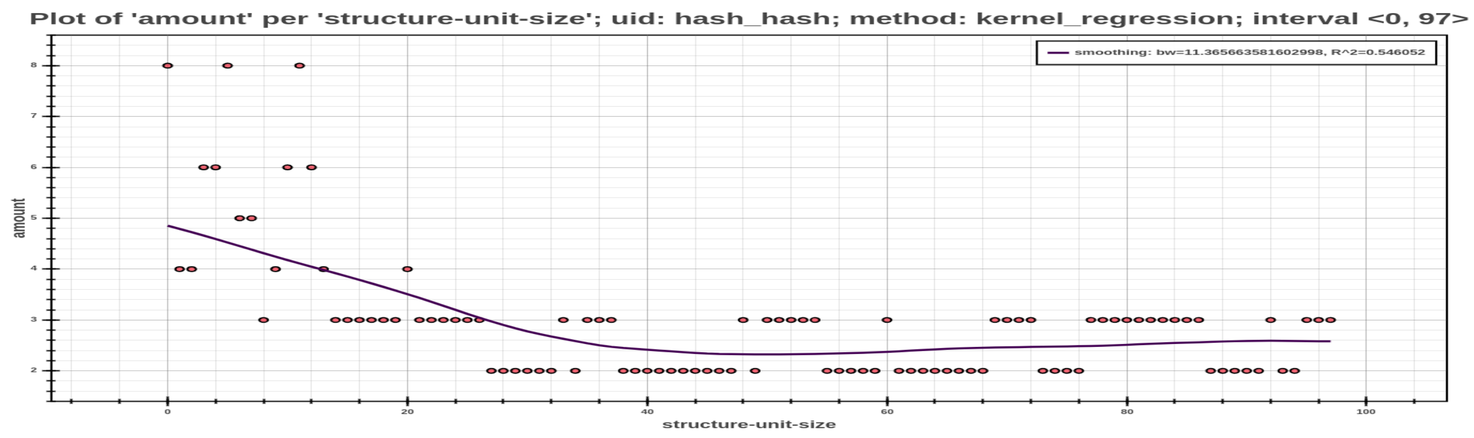

Figure 3. Example of kernel model created by kernel-regression post-processor with kernel-smoothing mode.

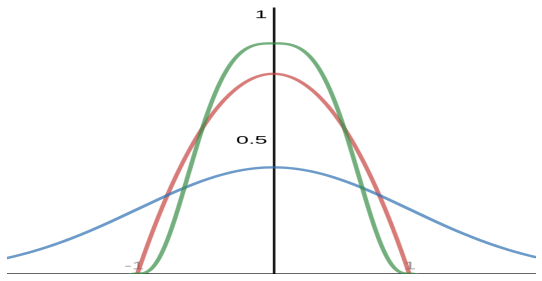

Figure 4. (—) Tricube kernel, (—) Gaussian Kernel, (—) Epanechnikov Kernel.

5.1 Smoothing Parameter Selection

The most critical factor of *Kernel regression* is the width of the *smoothing window*. This value significantly affects the smoothness of the resulting estimate, for example a large window width leads to overlay, and so to average data. When it comes to choosing a smoothing parameter, it is important to realise that the ultimate decision about the estimated curve is partially subjective, because even asymptotically optimised estimates contain a relatively large amount of noise [3].

One of the most widely popular and used methods to determine the optimum value of the smoothing parameter is the *cross-validation* method. This method is based on an estimate of the regression function, which omit the *i*-th observation.

$$\hat{f}_{-i}(x_i,h) = \sum_{\substack{l=1 \\ l\neq i}}^{n} W_l(x_i,h)Y_l, i = 1,\ldots,n. \tag{7}$$

The function of the cross-validation method can be defined as following:

$$CV(h) = \frac{1}{n}\sum_{i=1}^{n}(\hat{m}_{-i}(x_i,h) - Y_i)^2, \tag{8}$$

and estimation of the optimum value of the smoothing parameter is the point at which is set the minimum of this function, i.e.:

$$\hat{h}_{opt,0,k} = h_{CV} = arg \min_{h\in H_n} CV(h), \tag{9}$$

Except for this method, our post-processor supports other methods for optimal selection of smoothing parameter. *Akaike information criterion (AIC)* compares the quality of a set of kernel models with the different value of bandwidth to each other. The *AIC* creates several kernel models with various bandwidth and ranks them from best to worst. Except for these more complex methods, we implemented two simple rules to determine the smoothing parameter. *Scott's* and *Silverman's* rules were initially designed for density estimation but are usable for kernel regression too. Scott's rule of thumb produces a larger bandwidth, and therefore it is useful to estimate a gradual trend in a data-set. Except for all these options, a user can enter the numeric value of the smoothing parameter in all available modes of this post-processors.

6. Detection of Changes

The detection of changes is the last step in the current automatic process in our framework. The input of our methods is the pair of profiles, where the first is called the *baseline* profile and represents the stable base against which we compare the *target* profile, that represents a newly released version of the project. All methods in addition to the potential detected *performance change* provides an *error rate*, which denotes how significant a change has occurred in comparison to baseline and *confidence rate*, which can help to decide whether the changes is worthy of fixing. By providing these crucial aspects for each detected change, we achieve a high ratio of performance fixes.

The detection methods, which were described and evaluated in our previous work [2], were implemented primarily for models created by regression-analysis post-processor. The advantage of these methods is that besides the mentioned aspects it reports the *severity* of the changes, which is represent by individual kind of the model. Our new detection methods are able to work with all types of models. In particular, we propose two approaches first based on computing integrals and other on computing local statistics.

6.1 Integral Comparison

We based this simple heuristic on the assumption that the areas under the curves that represent the individual models, should be approximately equal. Therefore, the main idea of this approach is to compute the definite integral under the given curve. In the case of models created by regression-analysis an integral is computed from their formulae, which are represented by coefficients 2.1. In the case of non-parametric models an integral is computed from a pair of points, which represent the individual model. The computation is performed for every non-parametric model and for a best parametric model for every precise location (UID) from given pair of profiles.

$$\delta_x = \frac{\int_{x_{start}}^{x_{end}} f_t dx - \int_{x_{start}}^{x_{end}} f_b dx}{\int_{x_{start}}^{x_{end}} f_b dx}, \tag{10}$$

$$\delta = \begin{cases} \bigcirc & for\ |\delta_x| \leq \xi_\theta \\ \odot & for\ |\delta_x| \leq \xi_\Delta \\ \otimes & else \end{cases} \tag{11}$$

The determination of resulting changes is executing according to the value of relative error (δ_x) that is calculated from both values of integral, as it shows Equation 10. This value is subsequently compared with given thresholds and according to the result of this comparison is to determine changes. As shows Equation 11, there are two thresholds which serve for differentiating of three possible states — without change ($\bigcirc$), possible change ($\odot$), change ($\otimes$). The first threshold (ξ_θ) determines the boundary of change, which is accepted between both models and the second threshold (ξ_Δ) determines the uncertainty interval, in which may be potentially occurred change. The precise value of thresholds has been established based on experiments and on requirements of users, which does not accept errors with a small value of error rate.

In our case it is also sometimes appropriate to divide the data into more intervals and do a subsequent interval-based degradation detection. The partial results can give user precise locations, when the changes occur and therefore we decided implements another method adapted to these requirements. It is a method whose principle is the same as was described above, with only difference being that the integral is computed from the sub-intervals.

6.2 Local Statistics Metrics

This degradation method assumes the change of several statistics metrics on the individual sub-intervals. As in the first method, there is also the analysis executing on each non-parametric model and on the best parametric model for every precise location. The original values from each model are divided into several sub-intervals, wherein the number of sub-intervals and a minimal number of values in each sub-interval are predetermined. The exception is the last interval in which the number of points can be smaller. Subsequently, the following statistics metrics are calculated for each interval: *mean*, *median*, *maximum*, *minimum*, *sum*, *first* and *third quartile*.

After the computation of all these metrics for both baseline and target models we compare these statistics. It means that for each metric we compute a relative error against to baseline model and then according to its values we determine if change occurred. To report a change on a specific sub-intervals we must detected change in at least half of computed metrics. For an overall change on the whole interval an average value of the relative error must be computed through all sub-intervals, and higher than predetermined thresholds. The user receives information about the summary change represents by the mentioned average error rate (relative error), changes on the specific sub-intervals and the error rate on these intervals.

This method was implemented with usages of `NumPy` package and its modules. Thanks to manipulation with `NumPy` arrays and operating over all these arrays, we receive a several times faster calculation in comparison to usage of loops and lists.

7. Experimental Evaluation

We evaluated our new methods on repository of `vim` [5], which contains known performance issues. We tried to detect one known issue between two versions of `vim` as well as checking that two following versions had no significant changes. The issue, which is present in the version *v7.4.2293*, caused performance degradation of specific functions because it uses type `garray_T` instead of `hashtab_T` to collect tags. Since the tags are stored in a `garray_T`, `vim` has to perform a linear search of all existing tags every time a new tag is added. Using hash table (`hashtab_T`) obviously drastically improves the speed. This issue was fixed in *v8.0.0190* and therefore it will be our second tested version. We selected as the last version *v8.1.1005*.

We run *vim* with the following configuration: an argument `-u` with value `NONE`, which will ignore vim configure files `.vimrc`. As workload, we used the following command `'echo len(taglist("a"))'`, which will find all tags containing the letter *"a"* and at the end with command `q` we will terminates the editor.

[5] https://github.com/vim/vim

Table 1. Comparison of detection performance changes using *integral* method between different versions of *vim*.

		v8.0 - v7.4			v8.0. - v8.1		
	Model	-	+	?	-	+	?
#1	**RG**	18%	17%	13%	11%	8%	13%
	MA	5%	3%	0% (2)	0% (2)	1% (5)	0% (0)
	KR	6%	2%	0% (1)	0% (3)	1% (4)	0% (1)
#2	**RG**	17%	10%	10%	17%	6%	18%
	MA	6%	3%	0% (2)	0% (3)	1% (6)	0% (2)
	KR	8%	2%	0% (3)	1% (4)	1% (5)	0% (2)

Running *vim* with this configuration is complex and contains a large number of called functions, so we needed some level of abstraction for collecting reasonable data. Therefore, a *trace* collector was used to profiling this program as it provides the option of global sampling of calling each function and so we can monitor every n-th calling only. From collected profiles we subsequently create non-parametric models, by our three new post-processors, and then use detection methods to determine the possible changes. We tested each type of non-parametric models — regressogram (**RG**), moving average (**MA**), and kernel regression (**KR**) — and two detection methods, on two kinds of profiles with two values of the global sampling — 500 and 1000 — created over each traced version of *vim*.

Table 2. Comparison of detection analysis results by the *local statistics* method between different versions of *vim*.

		v8.0 - v7.4			v8.0. - v8.1		
	Model	-	+	?	-	+	?
#1	**RG**	11%	8%	13%	6% (26)	3% (14)	6% (25)
	MA	13%	8%	11%	6% (23)	3% (12)	6% (26)
	KR	12%	6%	11%	6% (24)	3% (12)	7% (29)
#2	**RG**	9%	5%	8%	8% (31)	3% (13)	14%
	MA	11%	4%	7%	5% (20)	1% (4)	12%
	KR	12%	5%	7%	5% (20)	2% (8)	13%

Tables 1 and 2 shows the results of our experiments. We compared two pairs of different versions, we chose version *v8.0.0190* as the *baseline* profile and remaining two versions as the *target* profiles. The different values of sampling are represented by rows *#1* and *#2* in the tables. Each row shows how much of functions (in percents) were reported as degradations (+) and optimizations (-). We use (*?*) to signify that some change was detected, but was not drastic enough to be reported.

In the first experiment (*v8.0 - v7.4*) we compared **333** common locations from both profiles and in the second case (*v8.0 - v8.1*) we compared in summary **388** location. We can see that in the second case we only detected minimal changes between specific models from each location of profiles, since the difference between these two versions should be stable. Most of the changes were reported using *regressogram* models, which confirms our assertion that it is an overestimating method. In first test case we confirmed the presence of issues [6] in *vim v7.4.2293*, according to our assumptions. The average increase of detected changes is equal to **7.00%**, which approximately corresponds to the impact of a known issue on specific measured functions. More thorough analysis of changes in specific functions has not been a part of these preliminary experiments, and will be subject of our future work.

8. Conclusion

We introduced new methods for modeling performance and detecting performance changes within the Perun [1] tool. Using our methods, we were able to detect one know performance issue in *vim* as well as comparing two other versions of the same project. Our future work will focus mainly on increasing the accuracy of our detection methods and improving the performance of post-processors for faster processing of collected profiles. Furthermore, we plan to evaluate our solution on other existing projects and potentially detect new unique performance changes.

Acknowledgements

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

References

[1] *Perun: Performance Version System*. https://github.com/tfiedor/perun. [Online; visited 15.3.2019].

[2] Pavela Jiří and Šimon Stupinský. *Towards the detection of performance degradation*. In *Excel@FIT'18*.

[3] Michael G Schimek. *Smoothing and regression: approaches, computation, and application*.

[4] Jiří Pavela. *Knihovna pro profilování datových struktur programů C/C++*. Master's thesis, BUT, FIT, 2017.

[5] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning*.

[6] Ján Koláček. *Jádrové odhady regresní funkce*. Dissertation's thesis, 2004.

[6] https://github.com/vim/vim/pull/1046