Run-time estimation for data processing tasks

Jan Bielecki

Abstract— **The aim of the research is to estimate the run-time of a data processing task (specific execution of a program or an algorithm) before the run. In the article we do not strictly focus on providing an estimation of maximum execution time (WCET[1]). We try to estimate the average-case execution time (ACET) which is absolutely enough for this use case requirements. The chosen approach is to create machine learning models using historical or testing data of the executions. Each execution of a program has the set of explanatory variables as properties of an input data and execution environment. Obviously, response variable is the run-time of a program.**

Keywords— **multicore computing, data processing, analysis of algorithms, run-time estimation, nonlinear regression, small-p regression, small-n regression, machine learning, KNN, SVR, RBF**

I. Introduction

A. Context

The research described in the article focuses on providing the tool for estimating the run-time of a single CAL module (data processing program or algorithm). Computation Application Language (CAL) is designed to write large-scale applications due to perform big data processing. Each CAL program consists of separately executed modules in sequence (with parallelization possibility) on virtual machines and sending the results further to the next module of the application. CAL language is a part of the system developed under the Baltic LSC[1] project.

During the data processing within the application run, each module is executed with some input data. It could be a different kind of data e.g. data frame or set of images. The execution of a module is invoked on a virtual machine with limited resources (RAM, mCPUs, GPUs). The properties of input data and the execution environment resources will be used by models to estimate the overall application execution time and price on a specific cluster.

Baltic Large Scale Computing (BalticLSC[2]) project is using CAL language to perform data processing tasks. Each task is an execution of a CAL application. CAL applications can use each module of the system in a reusable way. We can say that a single module is one block (commonly as a docker image) and an application is a schema of executing a sequence of blocks. Figure 1 shows the example application that consists of a few modules to perform face recognition on the input video data. Some sets of modules within a CAL application can be executed in parallel. Some modules can also be run as multiple instances of themselves to make data processing faster.

The demand for large-scale computation is tremendous and it is growing all the time. Mostly because of the high computational requirements of machine learning models and other tasks related to the Big Data thinks such as the Internet of Things [3]. To execute the computations, it takes a lot of hardware resources. Moreover, it is really frequent that data processing programs have common parts that could be shared between them to make the whole process of the program's engineering more granulated and universal. Focused on just one-step of processing, an engineer could provide a well optimized module that could be shared in many programs. The BalticLSC project's research focuses on providing the complete system, handling mentioned challenges of large-scale computing.

B. Issue

The worst-case complexity of an algorithm is the greatest number of operations needed to solve the problem over input data of a given size. The analysis of algorithmic complexity emerged as a scientific subject during the 1960s and has been quickly established as one of the most active fields of study[4]. The most common way to describe an algorithm complexity is the *big O* notation (collectively called Bachmann–Landau notation or asymptotic notation) which is the universal formula that describes how the run-time of an algorithm grows as the input data size grows.

In our work, we study the formula (more precise and complex one in comparison to the *big O* notation) that takes mixed sets of input data and run-time environment properties as arguments. Moreover, the models we introduce in further parts will provide the estimation of the data processing program run-time and not only the complexity formula as the *big O* notation does.

C. Related work

Since run-time estimation is a long-known issue, it is commonly studied from different perspectives and various approaches. There are many articles that, to some extent, coincides with the research carried out in this article. An overview of the problem with the list of possible, known solutions is described in the *estimation of the execution time in real-time systems*[5]. The authors focus on the *WCET* concept. This project aims not to estimate execution time as a deadline that could not be exceeded. In our work, we try to estimate the average-case execution time (*ACET*), which has a loser requirement for the final upper bound of the run-time than the *WCET*.

A similar approach to the run-time estimation have been introduce in the *Execution Time Analysis for Embedded Real-Time Systems*[6] - as is said in the section 5. of the article, a static timing anal-

[1]Worst-case execution time - the maximum amount of time that the execution can take.

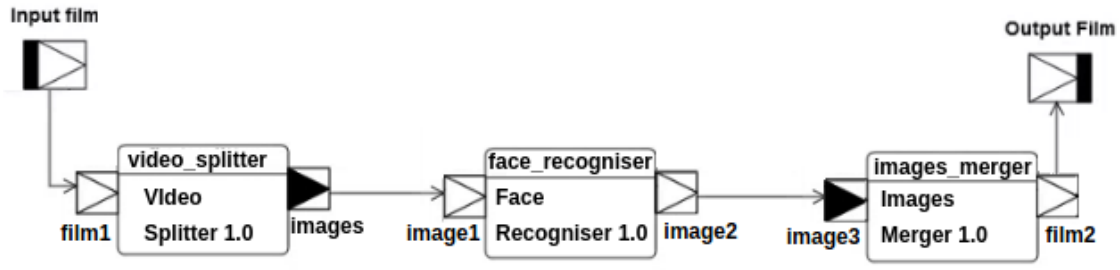


Fig. 1: *Face Recogniser* scheme. Application written in the CAL language.

ysis tool works by statically analysing the properties of the program (embedded to a specific structure like vector or graph) that affect its timing behavior. On the other hand, our approach is to statically analyze the properties of input data and the run-time environment resources. It allows to focus more on a single program and receive more accurate results. However, we have to provide a sufficient amount of input data for each module what is challenging. The fact that analysing programs run in BalticLSC system and they are reusable we assume that each module will be repeatedly triggered and it allows to carry enough data.

In section 3.6 of the *Using online worst-case execution time analysis and alternative tasks in real-time systems*[7] we can find a similar approach to carry out the regression using input data feature (section 3.6). The authors use an amount of image pixels as the only feature to estimate the execution time of a few different image processing algorithms. This is only a part of the whole approach for the worst-case execution time analysis described in the article, but it is the most related one to our work. It is worth noticing that there could be some specific input data properties for each algorithm that substantially impact the run-time. Models that we studied have general, strictly limited features, the same ones for each module, but one can extend it with some specific properties of a module.

To get a more precise run-time estimation results, one can study the program with similar input data. In the article *A Prediction Model for Measurement-Based Timing Analysis*[8] the authors made an experiment by generating random lists of data with the same properties and then train the artificial neural network to predict the *WCET*. They used *Gem5* to simulate the run-time environment. We have used the docker as an execution environment which allows receiving more real data with a natural noise introduce to the response variable (execution time). The authors develop their work in the next article[9] using the concept of surrogate models as a solution to the problem of generating training data. Such generating can increase overheads of the execution time estimation for processing algorithms with heavy input data. It is worth considering as an extension of our approach to generating some input data when the module is new and does not have any executions yet.

If we know that there will be not enough execution of each program within some system, we can create a single model for a whole system. The authors within the article *Nonlinear approach for estimating WCET during programming phase*[10] estimate a *WCET* by extracting the program features from the object code. It starts when the source code of a program is successfully compiled into object code. Then, the extracted features were used for subsequent sample optimization and *WCET* estimate. The authors use the SVR algorithm with RBF kelner as we did likewise. Having a set of features for each program, we can unify a model to a single instance and provide the features as a part of input data for the model. Another clue to the not enough amount of input data for a model is described in the article *A solution to minimum sample size for regressions*[11]. The authors explore the problem of a small amount of training data which also affects the result of our article. Testing data generation and simple algorithms for the run-time estimation are used to reduce the impact of a small n problem on the final results of our work.

D. Contribution

Despite the widely studied subject of estimating a program execution time, there is always a place to improve. Our work, presented in the article, contributes to the current knowledge of run-time estimation. We offer another perspective of solving the problem and one can find some valuable aspects of our work.

Most importantly, we used simple machine learning algorithms (*SVR* and *KNN*, more about the algorithms and how they fit the problem in the further sections) to estimate a program run-time. Creating a separate model for each module (program) instead of embedding the program structure to a vector of features or control flow graphs (CFGs) allows being more focused on each module separately. As input data features for our models, we determined the basic properties of a program's input data together with the properties of the run-time environment of an execution.

II. Data

We created a machine learning model for each module to estimate the run-time of a CAL module execution. We will get another data point to train our model with each module's run (within some application of the Baltic LSC system). The following

features (explanatory variables) make up the input data set for a model:

1. CPUs limit - called *mili-cores* - the fraction of a physical CPU used to carry out the module execution.
2. The total size of an input data in bytes.
3. The largest element size of an input data (if the input data is a set of files type, it is the largest size of a file within the set).
4. The average size of an input element.
5. The number of input data elements.

The last three features make clear sense only if input data is a set of files. Otherwise, if data is just a single file, the features can be a sort of data features representation carrying more detailed information about the data than only a total size. For the data frame type, the number of input elements can be equal to the number of columns. The max element size will be a quotient of the total size and the number of columns, and the average size will be equal to that quotient as well. In other cases (other kinds of input data), one can prepare specific, additional features and store their values for each execution (to enable their use for training in the future). In this work, we simplified the task to the features mentioned above. Our dependent value (that we are going to estimate) is an execution time of a module.

We created two CAL applications based on four modules with different types of input data. The first application (consisting of three modules) takes the movie as input data, marks people's faces on each frame, and then returns the movie with marked people's faces as output data. The second one consists of just a single module, and it searches for the best hyperparameters of the XGBoost algorithm within the parameters grid. Table II describes the modules that we used in the research. For each module, we created a set of 10 different input data sets. Next, we ran the modules with all the mentioned data sets using various CPU resources (from 0.5 CPUs up to 4.0 CPUs with 0.5 step). Finally, we received the data frame with the number of 4 (number of modules) * 8 (different CPUs resources) * 10 (number of input data sets) = 320 rows that we used to train and validate our models. Part of the models' input data is presented in Table I.

To make the models' data more understandable, we prepared two figures that show interactions between the features. Figure 2 shows how each column of the dataset is correlated with the other ones (green indicates positive correlation, red indicates negative correlation). As a correlation coefficient between two columns, we used Spearman's rank correlation coefficient. As we can see, the data-based input features are mostly positively correlated with each other to some point. They obviously are positively correlated with the run-time as well. The CPUs feature is negatively correlated with the run-time for each module (highest correlation for *face_recogniser*). As the data-based input features are strongly correlated, we performed the PCA analysis to determine how the specif column affects the input data variation. On the x-axis of Figure 3, starting from the left with the component with the highest impact on the data variance, we can see the PCA results. The colors on the Figure show the share of features in each of the components. CPUs' only environment-based feature is not correlated with other ones and impacts the data variation with a value equal to 1/(number of columns) = 0.2 for each module. The rest of the variance falls on data-based features. The PCA analysis shows that we could compress our input data to only three components and still keep almost all variance (the same situation for each module). We kept the original input data for more clearness, but one should consider dimensionality reduction if he has decided to provide many features.

III. Algorithms

A. Support Vector Regression

As a one of the machine learning algorithms to estimate time of a CAL module execution we will use *Epsilon-Support Vector Regression* [12] (SVR) - it is based on Support Vector Machine (SVM, orginally named *support-vector networks*[13]).

We will use the SVR model with an RBF kernel[14] which is the most known flexible kernel and it could project the features vectors into infinite dimensions. It uses Taylor expansion which is equivalent to use an infinite sum over the polynomial kernels. It allows modeling any function that is a sum of unknown degree polynomials.

Using kernel, the resulting algorithm is formally similar, except that every dot product is replaced by a nonlinear kernel function. This allows the algorithm to fit the maximum-margin hyperplane in a transformed feature space. The RBF kernel have the following form:

$$K_{RBF}(\vec{x}, \vec{x'}) = \exp\left(-\gamma||\vec{x} - \vec{x'}||\right) \text{ , where:}$$

- $||\vec{x} - \vec{x'}||$ is the squared Euclidean distance between the two vectors of features,
- γ - hyperparameter described in more details below.

In the SVR algorithm we are looking for a hyperplane y in the following form:

$$y = \vec{w}\vec{x} + b, \text{ where:}$$

- $\vec{x}$ - vector of features,
- $\vec{w}$ - normal vector to the hyperplane y, using a kernel, the $\vec{w}$ is also in the transformed space.

Training the original SVR means solving:

$$\frac{1}{2}||w||^2 + C\sum_{i}^{N}(\xi_i + \xi_i^*),$$

with the following constraints:

$$y_i - \vec{w}x_i - b \leq \epsilon + \xi_i$$

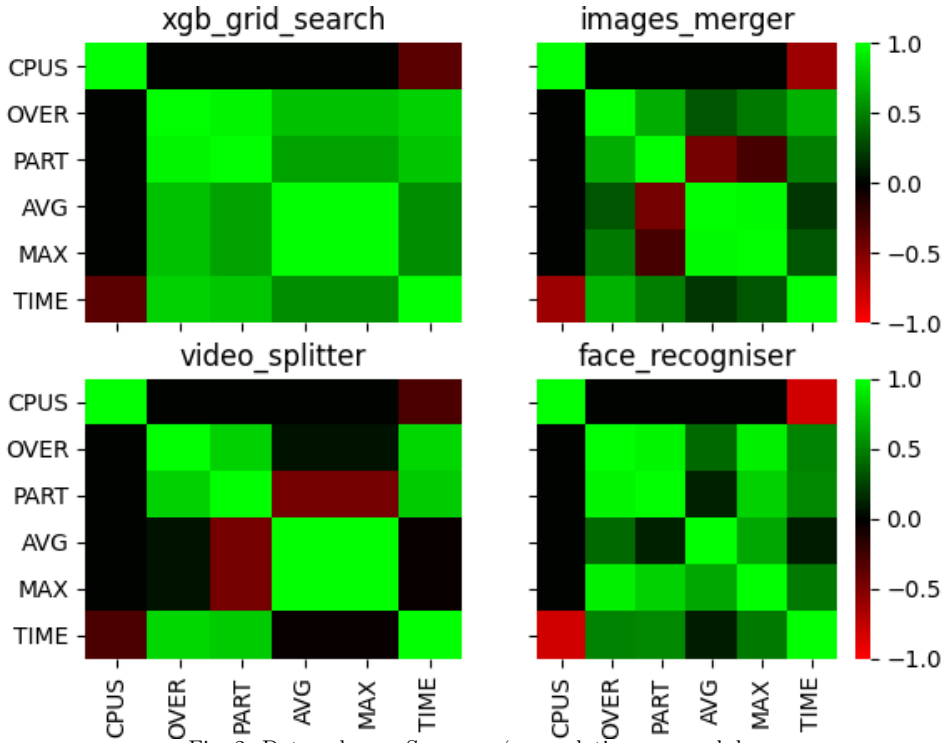


Fig. 2: Data columns Spearman's correlation per module.

Fig. 3: PCA importance per module.

Tabla I: Part of the data frame for models training and validation.

Module ID	mCPUs	total size [B]	number of elements	max size [B]	average size [B]	time [s]
1	4.0	1703379	544	3131	3131	3.909
1	4.0	809881	548	1477	1477	1.981
1	4.0	1711796	1392	1229	1229	11.371
...	...	...	...	...	...	...

$$-y_i + \vec{w}x_i + b \leq \epsilon + \xi_i^*$$

$$\xi_i \xi_i^* \geq 0$$

Figure 4 shows the example of a hyperplane with marked ξ and ϵ parameters. As we have already chosen the RBF kernel for the SVR algorithm, our modeling is simplified just to find the best values of the following hyperparameters[15]:

1. C -the weight of an error cost. The regulariza-

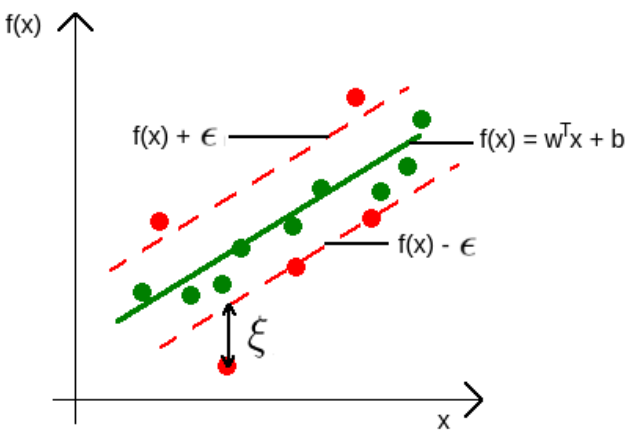
Tabla II: Modules that we used in the research.

ID (APP ID)	Name	Input data
1(1)	video_splitter	video file
2(1)	face_recogniser	image files
3(1)	images_merger	image files
4(2)	xgb_grid_search	CSV file

tion[2] hyperparameters have to be strictly positive. The example from the figure use l1 penalty (the used modeling library uses the squared epsilon-insensitive loss with l2 penalty). The strength of the regularization is inversely proportional to C. The larger value of C, the more variance is introduced into the model.

2. *epsilon* - It specifies the epsilon-tube within which no penalty is associated in the training loss function with points predicted within a distance epsilon from the actual value. As it is shown of Figure 4 the green data points do not provide any penalty to the loss function because they are within the allowed epsilon range around the approximation[3].
3. *gamma* - The *gamma* hyperparameter can be seen as the inverse of the radius of influence of samples selected by the model as support vectors. Increasing the value of *gamma* hyperparameter causes the variance increase what is shown in the figure 5[3].

Fig. 4: Visualization of the SVR's epsilon and gamma parameters [?].



You can find detailed information about C and gamma params in sklearn library documentation[16].

B. K-nearest Neighbors Regression

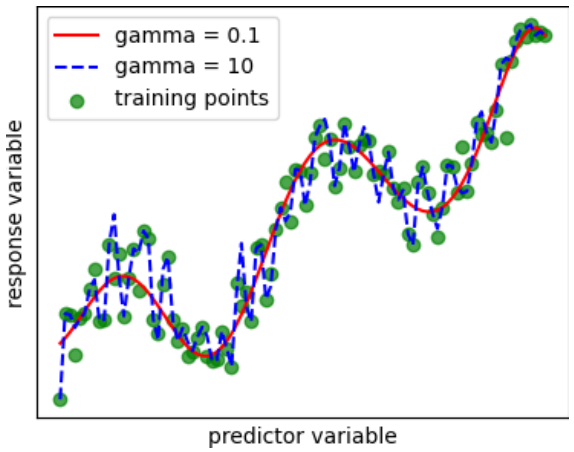
Another, examined algorithm used to estimating time of a module execution is a simple *KNN*[4] regression. The target is predicted by local interpolation of the targets associated with the nearest neighbors in the training set [17].In other words, the target assigned to a query point is computed based on the mean of the targets of its nearest neighbors.

[2]regularization is a way to give a penalty to certain models (usually overly complex ones)

[3]The figure is based on some example data and it only shows the hyperparameter influence on model.

[4]*KNN* - k-nearest neighbors

Fig. 5: SVR's gamma hyperparameter [?] influence on the model variance.



To use the algorithm, we have to define values of the following parameters:

1. k - number of the nearest neighbors
2. *weights* - weight function used in prediction. The basic nearest neighbors regression uses uniform weights: that is, each point in the local neighborhood contributes uniformly to the classification of a query point. Under some circumstances, it can be advantageous to weight points such that nearby points contribute more to the regression than faraway points. The weights can be calculated from the distances using any function the linear one, for example.
3. *algorithm* - the procedure to calculate k-nearest neighbors for the query point. It does not have a direct impact to the final regression result, but the parameters of the algorithm surely have. For example, using the BallTree algorithm, we have to choose the *metric* parameter that will be used to calculate the distance between data points. It could be, for example, the Minkowski [18] metric with the l2 (standard Euclidean [19]) distance metric or any function that will calculate the distance between two points.

C. Perceptions

The problem we are trying to solve is a small-p regression type. Our models are based on only a few input features, so the examined space is low-dimensional. We do not need complex machine learning tools like XGBoost [20] or neural networks to receive satisfactory results. The algorithms described above was selected for modeling because of their simplicity.

We assume that most of the modules have polynomial time complexity. The *SVR* is a natural choice for the type of problem we are trying to solve because of its ability to model any function that is a combination of any polynomials. On the other hand, we decided to use the *KNN* as well.

The simplest way to predict a module run-time is to look at the historical executions of a module with the input data that have similar properties to the executing one. That is what the *KNN* algorithm

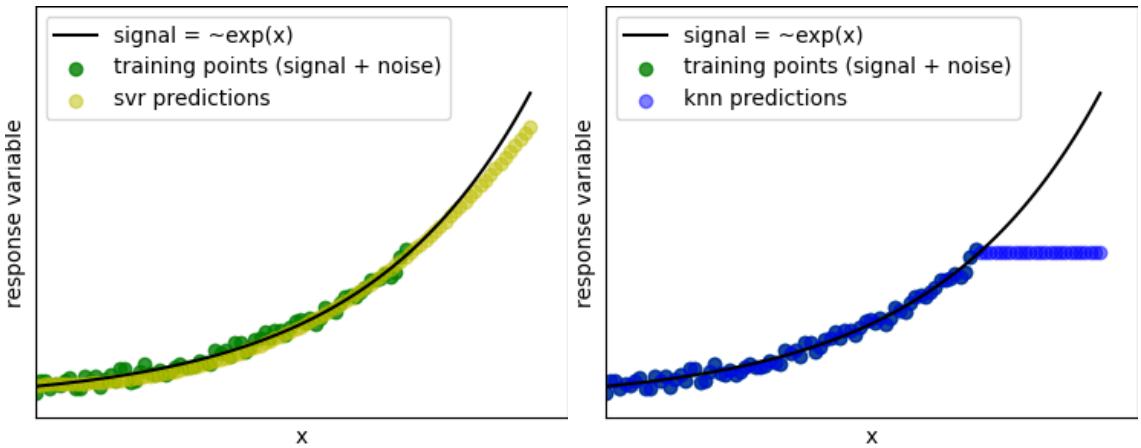


Fig. 6: Regression on the out of training range data.

does. A huge disadvantage of the algorithm is a lack of ability to provide acceptable estimations for outstanding data points from the training data points. The problem is visualized in the Figure 6. It presents the regression on the example input data. The *KNN* cannot predict the outstanding samples correctly.

IV. Training and validation

A. Training

The training pipeline for each module contains the following steps:

1. Having the 80 data points, we divide them into training and test data sets with 1:3 proportion. We freeze the test data set to use it only for the validation of final models.
2. Standardization of a data set is a common requirement for many machine learning estimators: they might behave badly if the individual features do not more or less look like standard normally distributed data (e.g. Gaussian with 0 mean and unit variance)[21]. We scale each column (feature) of the data set using the following formula:

$$\vec{x}'_f = \frac{\vec{x}_f - \mu_f}{\sigma_f}, \text{ where:}$$

   - $\vec{x}_f$ - data set column *f*,
   - μ_f - mean of the column *f*,
   - σ_f - standard deviation of the column *f*.
3. Each algorithm have a few parameters that should be chosen wisely to achieve better results. It is hard to predict the best value of a continuous parameter. What we did is an exhaustive search over specified parameter values for an estimator from the given possible values. For each combination of the parameter values we validate the model using 5-fold cross validation on the training data set. It is called a *grid search*[22].
4. Finally, we retrained our model using the entire training data set and the parameters that have been found in the previous step.

The previous section contains the description of model parameters. The parameters grid for the *SVR* algorithm is listed below:

```
'gamma': [0.0001, 0.0002, 0.0004, 0.0008,
0.0016, 0.0032, 0.0064, 0.0128],
'epsilon': [1e-06, 2e-06, 4e-06, 8e-06,
1.6e-05, 3.2e-05, 6.4e-05, 0.000128,
0.000256, 0.000512, 0.001024],
'C': [1000.0, 2000.0, 4000.0, 8000.0,
16000.0, 32000.0, 64000.0, 128000.0,
256000.0, 512000.0, 1024000.0, 2048000.0]
```

The following listening contains the chosen parameters grid for the *KNN* algorithm:

```
'n_neighbors': [1, 2, 3, 4, 5, 6, 7, 8,
9, 10, 11],
'weights': ['uniform', 'distance'],
'p': [1, 2]
```

The best model parameters based on the coefficient of determination R^2 of the prediction are presented in Table III.

We assume that each BalticLSC module will be executed many times. These assumptions induce many training data points for each model. However, there will be diversities between the number of executions per module. Moreover, the run-time of new modules cannot be predicted due to the lack of a training set. To investigate how the number of training samples affects the relative error of the regression, we calculated the learning curve. The results are shown in Figure 7. In line with what could be expected, the relative error decreases (with some hesitations) as the amount of training data increases for each module.

B. Validation

Since our models were trained only on the training data sets, we had the possibility to use test data set as a completely new data to validate our models. The separated test data guarantee that the validation results will not be distorted. Training with the cross-validation secures the model from over-fitting and make it able to generalize more. Table III shows the final results of estimation error for each module and algorithm. The error was calculated using the belowe formula:

$$error = \frac{1}{N}\sum_{i}^{N}\frac{|y_i - \bar{y_i}|}{y_i},$$

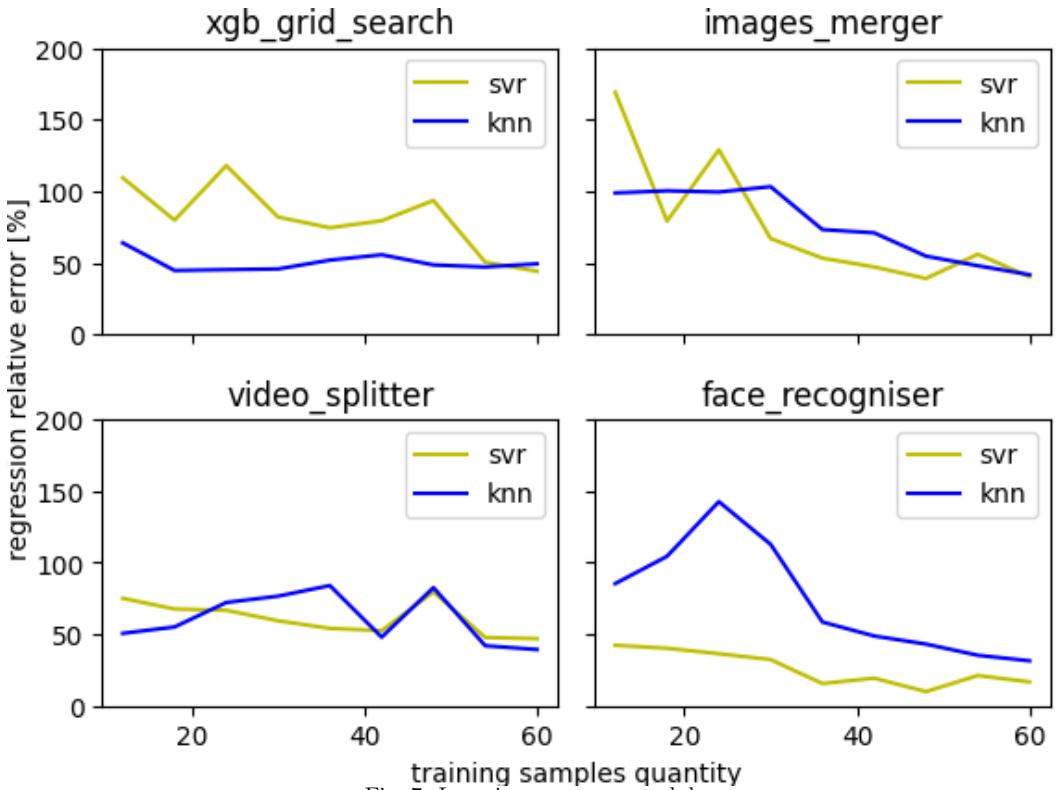


Fig. 7: Learning curve per module.

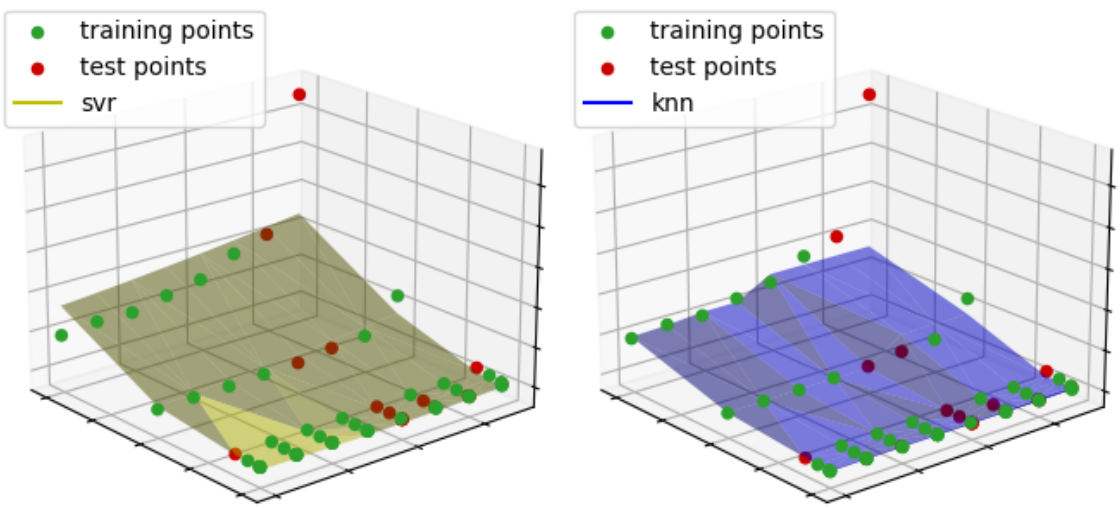


Fig. 8: Regression surfaces for *xgb_grid_search* module.

Tabla III: The final results.

Algorithm:	SVR		KNN	
	best params	relative error [%]	best params	relative error [%]
video_splitter	C: 512e+03, γ: 8e-04, ϵ: 3.2e-05	46.9	n: 2, *weights*: *distance*, p: 1	39.3
face_recogniser	C: 64e+03, γ: 3.2e-05, ϵ: 8e-06	16.7	n: 1, *weights*: *uniform*, p: 1	31.4
xgb_grid_search	C: 256000.0, γ: 1e-04, ϵ: 2e-06	44.7	n: 2, *weights*: *uniform*, p: 1	49.3
images_merger	C: 512e+03, γ: 1.6e-03, ϵ: 8e-06	40.6	n: 2, *weights*: *distance*, p: 1	41.7

where:
N - number of test data point = 20,
y - original run-time,
$\bar{y}_i$ - predicted run-time.

The Figures from 8 to 11 shows surfaces of regression for each module and algorithm. One can see that the *KNN* algorithm created the more irregular surface than *SVR* which gives a plus in the form of better generalization for the latter. The axes are not labeled on the plots to make the figures more transparent. The z-axis represents the run-time. The one on the left side of the floor is the overall size of the input data. The axis on the right side of the floor represents the number of CPU fractions. Besides the surface, the figures show training and test data points as accordingly green and red dots. The

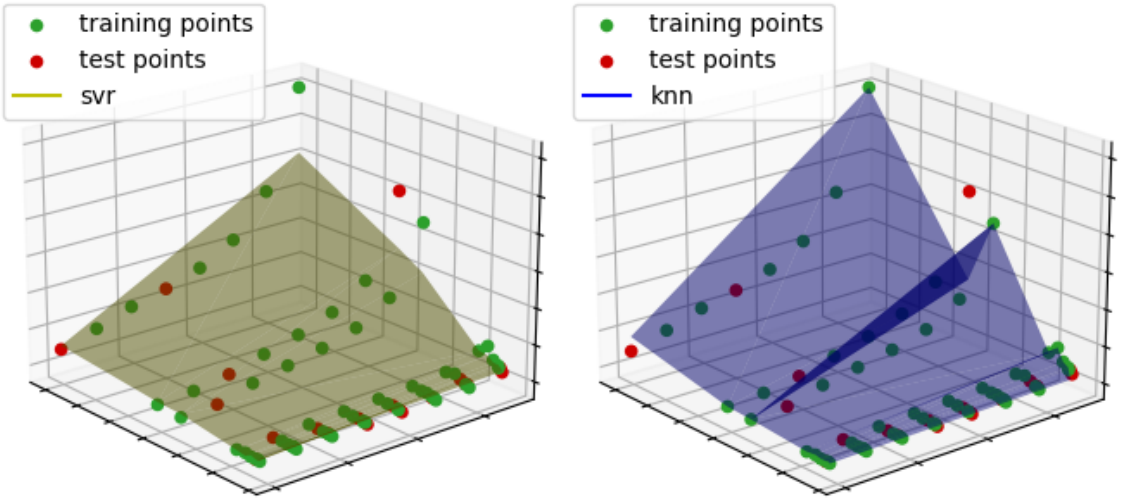


Fig. 9: Regression surfaces for *images_merger* module.

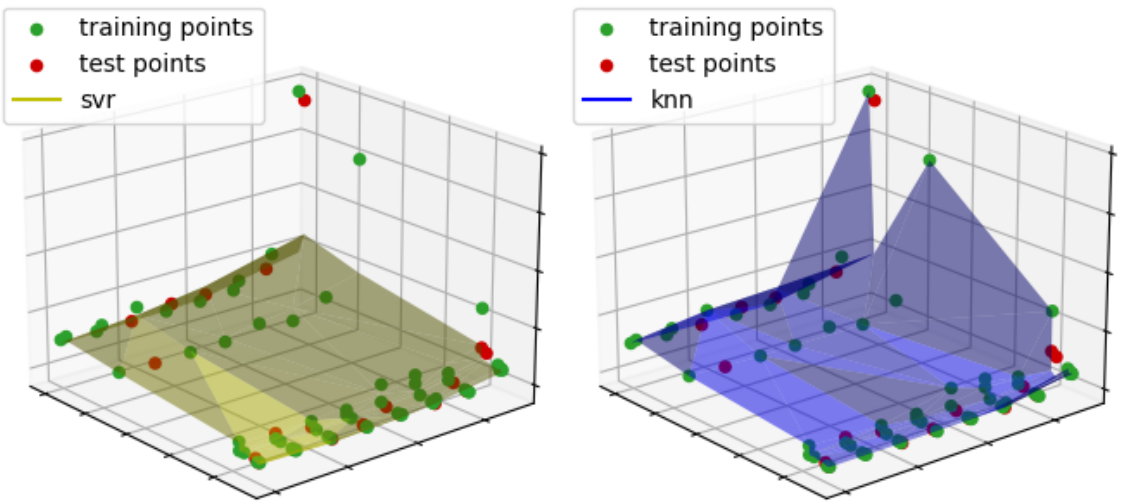


Fig. 10: Regression surfaces for *video_splitter* module.

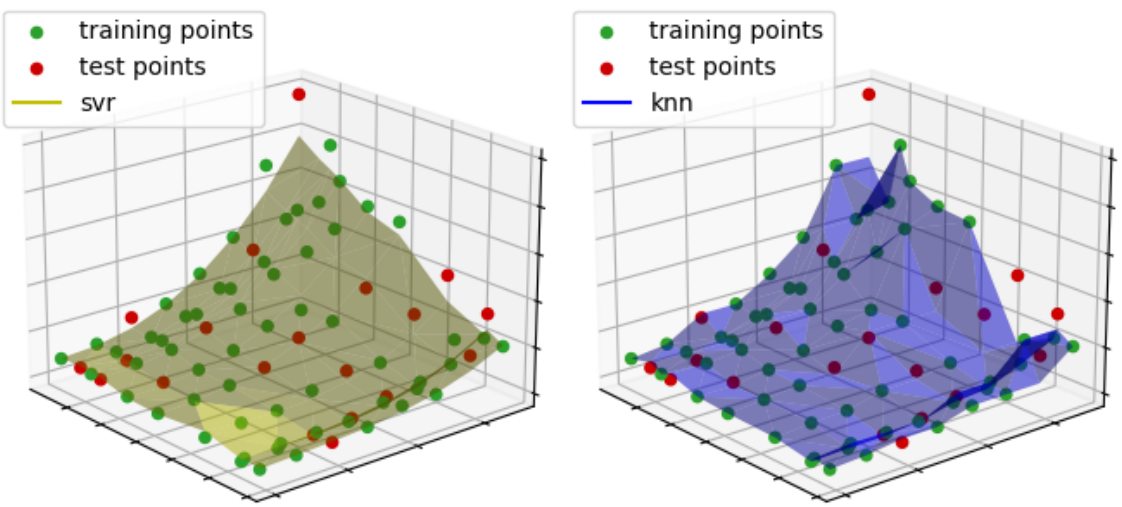


Fig. 11: Regression surfaces for *face_recogniser* module.

input data set was spitted randomly for each module in the same way for each algorithm. One can see that the data points for the *face_recogniser* module have a more uniform distribution for the overall size feature. It leads to a more smooth regression surface and smaller relative error (see Table III).

V. Conclusions

As the *PCA* analysis shows (see the Figure 3 and its description), the input data for our models have approximately three independent components that contain all variations of the data. This means that by using these components, we will reduce the model to a three-dimensional feature space only. We did not decide to use reduced space, but one can use the reduction to the more complex case because each model can be extended with additional features based on the specific module. For example, there could be some additional parameters required to run a module, and the parameters have an impact on the execution time. These parameters should be included in the set of input features for the module‘s model.

Analyzing only the final results, one can see that the used algorithms give relatively similar results with a bit of advantage for the *SVR* (average error of 38.0%, comparing to 40.4% for the *KNN*). The more critical characteristic which makes the *SVR* model more promising in use is the possibility to estimate the run-time in the more general way for the inputs out from the training range (see Figure 6 and its description).

In the subject of estimating the execution time of the whole application, the work that was done is not enough. It has to be noticed that we must also predict the input data properties for the next modules in the queue because, at the start, we only know the input data properties for the starting module of the application. This is true for applications consisting of more than one module. It will be implemented in further work to make the possibility to estimate the time of whole application execution.

References

[1] Baltic Sea region, *BalticLSC: main webpage of the project*, `https://www.balticlsc.eu`, visited 14.03.2021.

[2] Kamil Rybińsk Radosław Roszczyk Agris Šostaks, Michał Śmiałek, *BalticLSC: BalticLSC Admin Tool Technical Documentation, Design of the Computation Application Language, Design of the BalticLSC Computation Tool*, `https://www.balticlsc.eu/wp-content/uploads/2020/03/O5.2A_O5.3A_O5.4ABalticLSC_Software_Design.pdf`, 14.03.2021.

[3] Flüchter K. Wortmann, F., *Internet of Things*, Bus Inf Syst Eng 57, 221–224, 27 March 2015.

[4] Audrey A. Nasar, *The history of Algorithmic complexity*, The Mathematics Enthusiast: Vol. 13 : No. 3 , Article 4, 2016.

[5] V. P. Kozyrev, *Estimation of the execution time in real-time systems*, Program Comput Soft 42, 41–48, 22 January 2016.

[6] Jakob Engblom Andreas Ermedahl, *Execution Time Analysis for Embedded Real-Time Systems*, CRC Press, January 2001.

[7] Fredrik Bakkevig Haugli, *Using online worst-case execution time analysis and alternative tasks in real time systems*, Norwegian University of Science and Technology, June 2014.

[8] Muhammad Arif Syed Abdul Baqi Shah, Muhammad Rashid, *A Prediction Model for Measurement-Based Timing Analysis*, ICSCA '17, February 2017.

[9] Muhammad Arif Muhammad Kashif Muhammad Rashid, Syed Abdul Baqi Shah, *Determination of Worst-Case Data Using an Adaptive Surrogate Model for Real-Time System*, Journal of Circuits, Systems and Computers 29(1), January 2019.

[10] Zhaoyang Qu Fanqi Meng, Xiaohong Su, *Nonlinear approach for estimating WCET during programming phase*, Springer Science+Business Media New York, 22 July 2016.

[11] Pedro F. Quintana-Ascencio David G. Jenkins, *A solution to minimum sample size for regressions*, Plos One, February 21, 2020.

[12] Dr. Saed Sayad, *Support Vector Regression*, `https://www.saedsayad.com/`, visited 17.01.2021.

[13] Vladimir Vapnik Corinna Cortes *Support-Vector Networks*, Machine Learning volume 20, pages 273–297, 1995.

[14] Wikipedia, *Radial basis function kernel*, https://en.wikipedia.org/wiki/Radial_basis_function, visited 03.03.2021.

[15] David Cournapeau et al., *Support Vector Regression*, sklearn, visited 27.12.2020.

[16] David Cournapeau et al., *RBF parameters*, sklearn, since 2007.

[17] David Cournapeau et al., *KNN Regression*, sklearn, visited 10.04.2021.

[18] wikipedia.org, *Minkowski distance*, `https://en.wikipedia.org/wiki/Minkowski_distance`, visited 10.04.2021.

[19] wikipedia.org, *Euclidean distance*, https://en.wikipedia.org/wiki/Euclidean_distance, visited 10.04.2021.

[20] Guestrin C. Chen, T., *XGBoost: A Scalable Tree Boosting System*, Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (pp. 785–794), 2016.

[21] David Cournapeau et al., *Standard scaler*, scikit-learn.org, visited 18.04.2021.

[22] David Cournapeau et al., *Grid search*, scikit-learn.org, visited 18.04.2021.