Run-time estimation for data processing tasks

Statistic under AI and its application to engineering sciences

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 is focus 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 application in due to perform big data processing. Each CAL program consists of modules that are separetely executed in sequence (with parallelization possibility) on virtual machines and sending the results futher to a 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 different kind of data e.g. data frame or set of images. The execution of a module is invoke on a virtual machine with limited resources (RAM, mCPUs, GPUs). The properties of an input data and the execution environment resources will be used 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. Each module can be used in many CAL applications. We can say that a single module is a one block (commonly as a docker image) and an application is the schema of executing a sequence of blocks. Figure 1 shows the example application that consists of a few modules to perform a face recognition on the input video data. Some set of modules within a CAL application can be executed in parallel. Some modules can be also run as a multiple instances of itself to make data processing faster.

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

The worst-case complexity of an algorithm is the greatest number of operations needed to solve the problem over an input data of a given size. The analysis of algorithmic complexity emerged as a scientific subject during the 1960's and has been quickly established as one of the most active fields of study[20]. 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 describe how the run-time of an algorithm grow 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 it the *big O* notation does.

B. Related work

The articles listed above contain content that to some extent coincides with the research carried out in this article:

- *Estimation of the execution time in real-time systems*[3] - an overview on the different approaches to estimate the program execution time. The authors focus on the *WCET* concept. The aim for this project is not to estimate execution time as a deadline that could not be exceed. We try to estimate the average-case execution time (*ACET*).
- *Execution Time Analysis for Embedded Real-Time Systems*[4] - as is said in the section 5. of the article, a static timing analysis tool works by statically analysing the properties of the program (embedded to a specific structure like vector or graph) that affect its timing behavior. Our approach, on the other hand, is to statically analyze the properties of input data and the run-time environment resources.
- *Using online worst-case execution time analysis and alternative tasks in real time systems*[5] - 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 one feature to estimate the execution time of few different image processing algorithm.
- *A Prediction Model for Measurement-Based Timing Analysis*[6] - the authors made en experiment by generating random lists of data with the same properties and then train the artificial neural network to predict the *WCET*. They

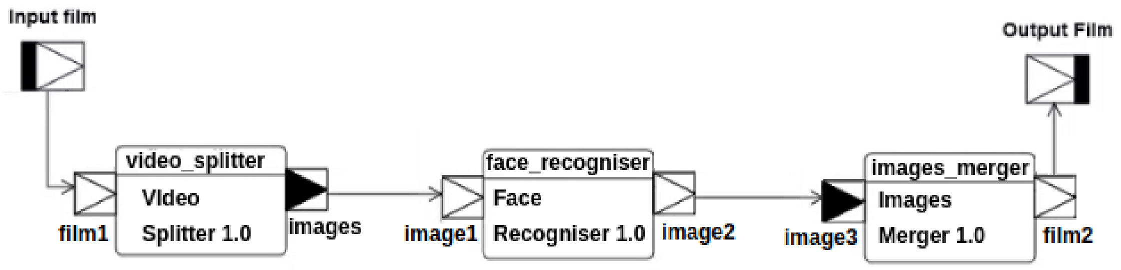


Fig. 1: *Face Recogniser* scheme. Application written in the CAL language.

used *Gem5* to simulate the run-time environment. We have used the docker as an execution environment which allows to receive more real data with a natural noise introduce to the response variable (execution time). The authors develop their work in the next article[7] using the concept of surogate 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.

- *Nonlinear approach for estimating WCET during programming phase*[8] - the authors estimate a *WCET* by extracting the program features from object code. It starts when the source code of a program is successfully compiled into object code. Then, the extracted features was used for subsequent samples optimization and *WCET* estimate. The authors use SVR algorithm with RBF kelner as we did likewise.
- *A solution to minimum sample size for regressions*[9] - 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 algorithm for the run-time estimation are used to reduce the impact of a small n problem on the final results of our work.

C. Contribution

Using simple machine learning algorithms (*SVR* and *KNN*, more about the algorithms and how they fit to the problem in the further sections) to estimate a program run-time.

Determine the basic properties of an input data and run-time environment for data processing task.

Creating a separate model for each module (program) instead of embedding the program structure to a vector of features or control flow graphs (CFGs).

II. Data

The figure 2 shows how each column of the data set is correlated with the other ones. As a correlation coefficient between two columns we used *Spearman*'s rank correlation coefficient.

The figure 3 shows ...

To estimate time of a CAL module execution we create a machine learning model for each module. With each execution of the module within some application of the Baltic LSC system, we will get another data point to train our model. We will use the following features (explanatory variables)[2] as an input data for a model:

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

This last three features make clear sense only if an 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 then only a total size. For the data frame the number of input elements can be eqaul 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.

Obviously, our dependent value (that we are going to estimate) is an execution time of a module.

We create two CAL applications based on 4 modules with a different types of an input data. The first application, consisting of 3 modules, take the movie as an input data, marks faces of people on each frame and return the movie with marked people faces as an output data. The second one consist of just a single one module and it do a search of the best hyperparameters of XGBoost algorithm within parameters grid. Table I describe the modules that we used in the research.

Tabla I: 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

For each module we created set of 10 different input data. Next, we ran the modules with the mentioned datas using different mCPUs resources (from

[2]As a docker container that will be used to execute a module do not use SWAP memory, RAM is not colerated with execution time. In this program we will not use modules with GPU support. Summarazing, the GPUs and RAM resources limits are not taken into account for modeling.

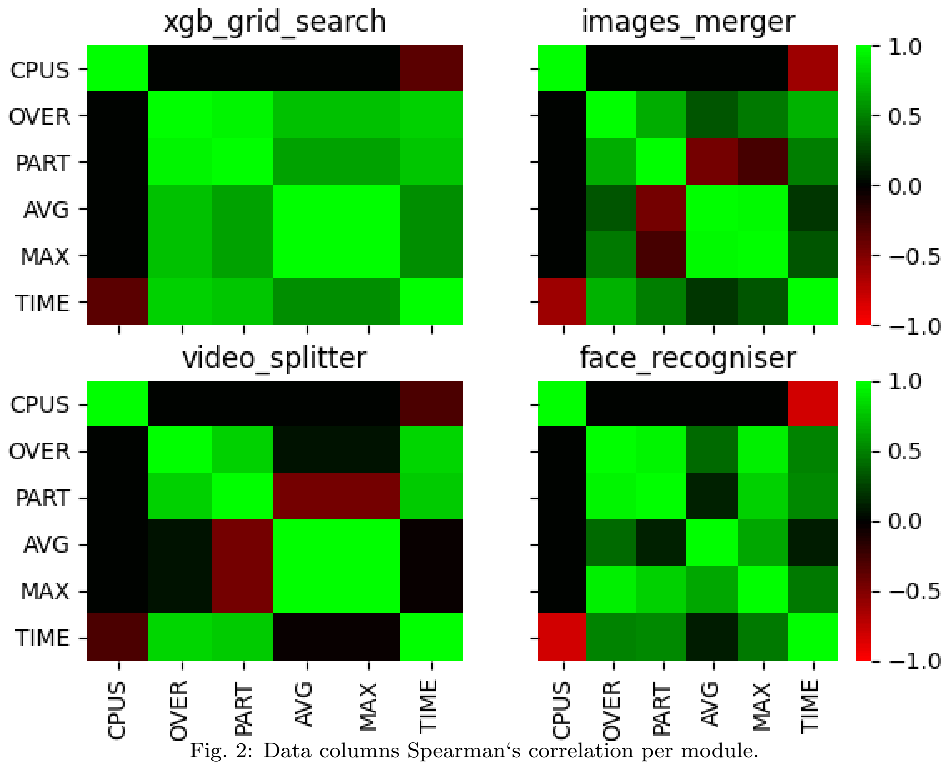


Fig. 2: Data columns Spearman‘s correlation per module.

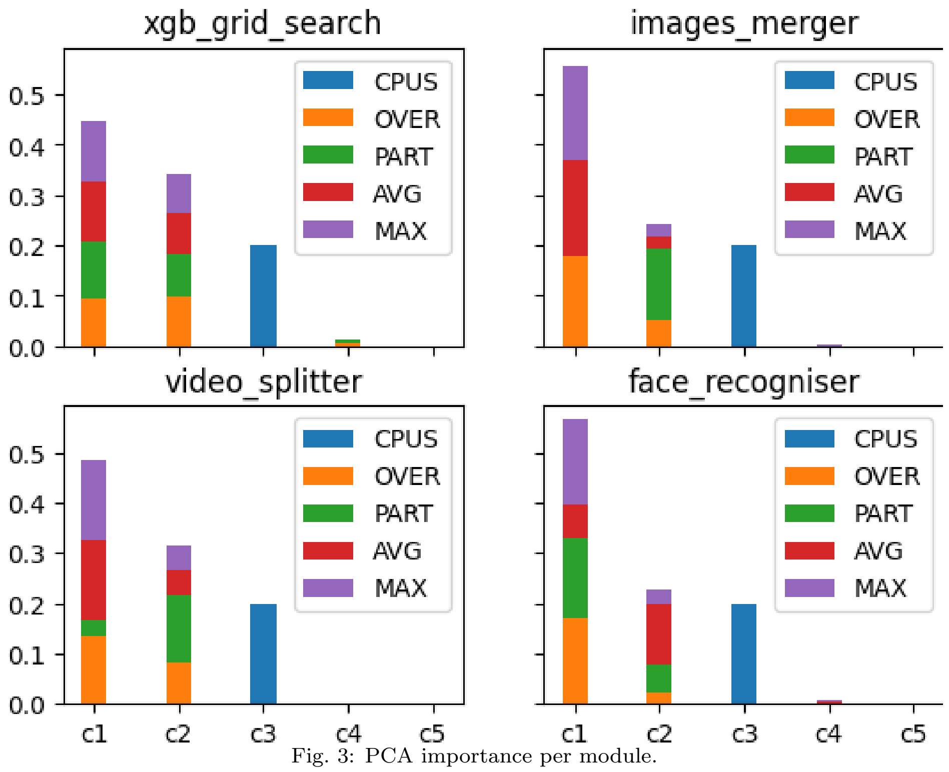


Fig. 3: PCA importance per module.

0.5 mCPUs up to 4.0 mCPUs with 0.5 step).

Finally, we received the data frame with the number of 4 (number of modules) * 8 (different mCPUs resources) * 10 (number of input data sets) = 320 rows that will be used to train and validate our models which part of is presented in the table III.

III. Algorithms

A. Support Vector Regression

As a one of machine learning algorithms to estimate time of a CAL module execution we will use *Epsilon-Support Vector Regression* [10] (SVR) - it is based on Support Vector Machine (SVM, orginally named *support-vector networks*[11]).

We will use SVR model with a RBF kernel[12] 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 to model any function that is a sum of unknown degree polynomials.

Using kernel, the resulting algorithm is formally

Tabla II: 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
...	...	...	...	...	...	...

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:

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

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

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

Figure **??** shows the wanted hyperplane with marked ξ and ϵ parameters. As we already choosed the RBF kernel for the SVR algorithm, our modelling is simplified to just find the best values of the following hyperparameters[13]:

1. C -the weight of an error cost. The regularization[3] hyperparameer, have to be strictly positive. The example from the figure use l1 penalty (the library that we use to modelling use 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 the 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[4].
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[4].

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

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

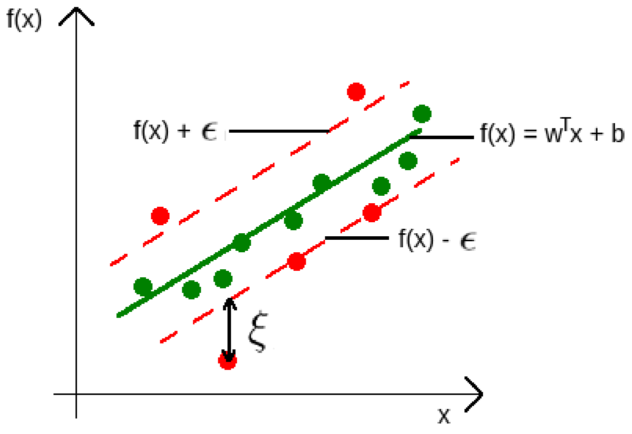
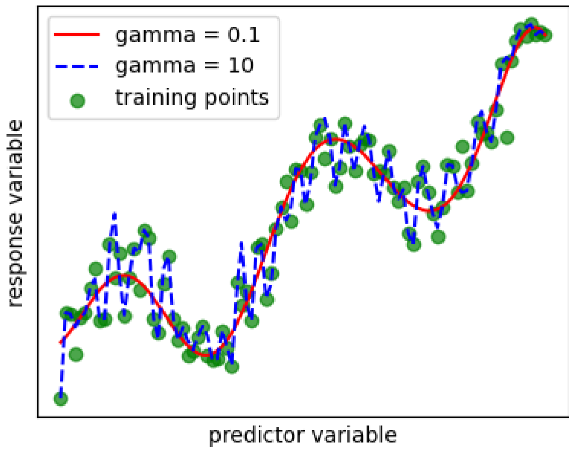


Fig. 5: SVR's gamma hyperparameter [?] influence on the model variance.



You can find detailed information about C and gamma params in sklearn library documentation[14].

B. K-nearest Neighbors Regression

Another, examined algorithm used to estimating time of a module execution is a simple *KNN*[5] regression. Using the algorithm, a target is predicted by local interpolation of the targets associated of the

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

[5]*KNN* - k-nearest neighbors

nearest neighbors in the training set [15].In another words, the target assigned to a query point is computed based on the mean of the targets of its nearest neighbors.

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 for example the linear one.
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 [16] metric with the l2 (standard Euclidean [17]) distance metric or any function that will calculate the distance between two points.

IV. Training and validation

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 1:4 proportion.
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)[18]. 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 in order to the 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 (as it was mentioned in the first step). It is called a *grid search*[19].
4. Finally, we retrained our model using the full data set and the parameters that were found in the previous step.

A. Support Vector Regression

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]
```

B. K-nearest Neighbors

The parameters grid for the *KNN* algorithm is listed below:

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

C. Final results

D. Training

...

E. Validation

...

V. Conclusions

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 the impact on the execution time. This paramaters should be included in the set of input features for the module‘s model.

`https://github.com/rasenjop/Plantilla-Jornadas-Sarteco`.

Thanks

Thanks for the help and support ...

Referencias

[1] FILL IT, *BalticLSC: main webpage of the project*, FILL IT, 14.03.2021.
[2] FILL IT, *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/2020/03/05.2A_05.3A_05.4ABalticLSC_Software_Design.pdf*, FILL IT, 14.03.2021.
[3] “Estimation of the execution time in real-time systems,” .
[4] “Execution time analysis for embedded real-time systems,” .
[5] “Using online worst-case execution time analysis and alternative tasks in real time systems,” .
[6] “A prediction model for measurement-based timing analysis,” .
[7] “Determination of worst-case data using an adaptive surrogate model for real-time system,” .
[8] “Nonlinear approach for estimating wcet during programming phase,” .
[9] “A solution to minimum sample size for regressions,” .
[10] Dr. Saed Sayad, *Support Vector Regression*, FILL IT, 17.01.2021.

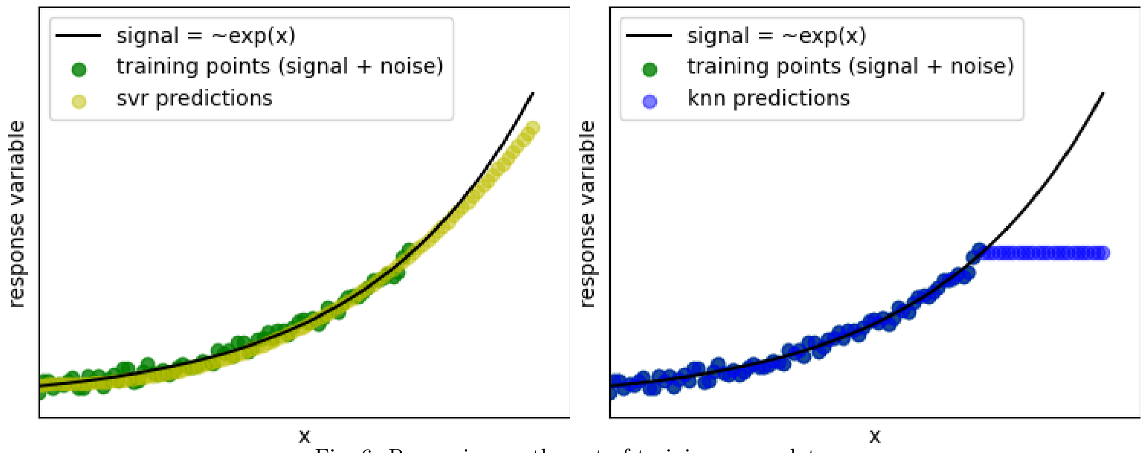


Fig. 6: Regression on the out of training range data.

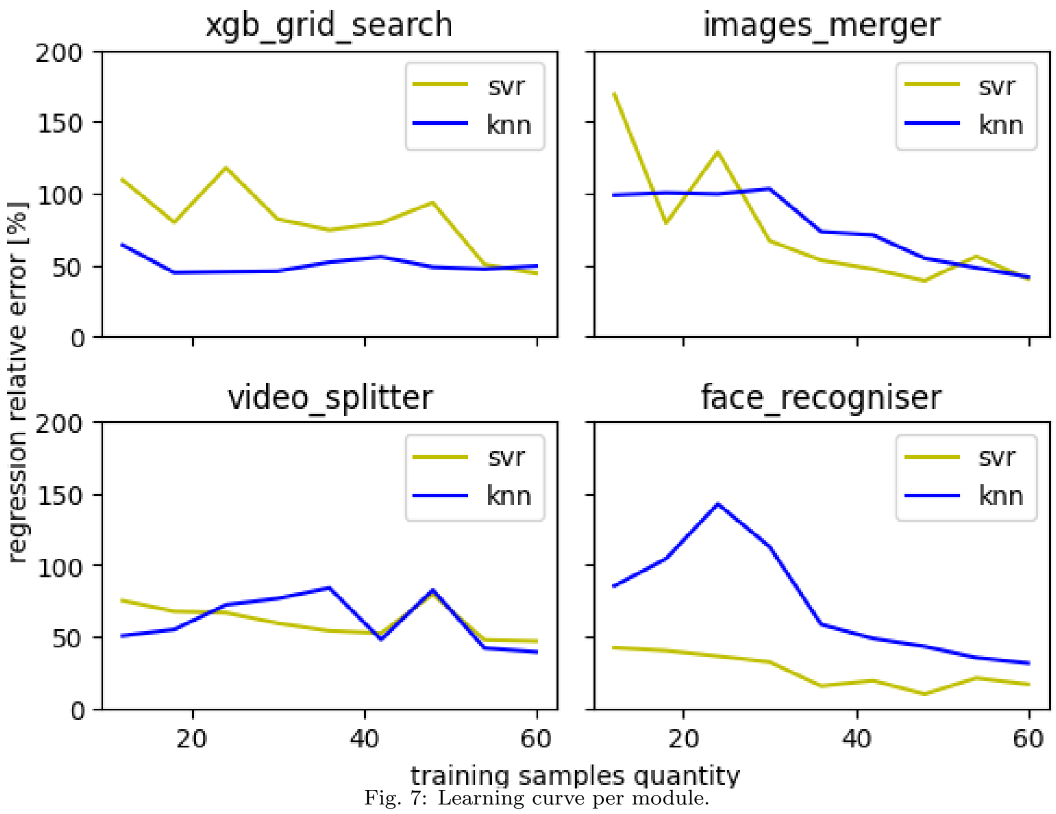


Fig. 7: Learning curve per module.

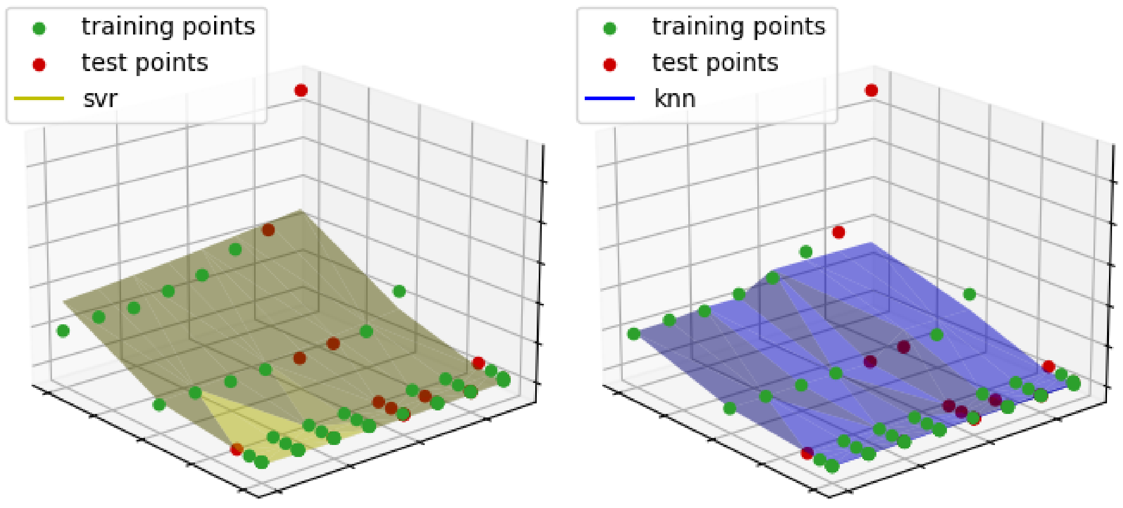


Fig. 8: Regression surfaces for *xgb_grid_search* module.

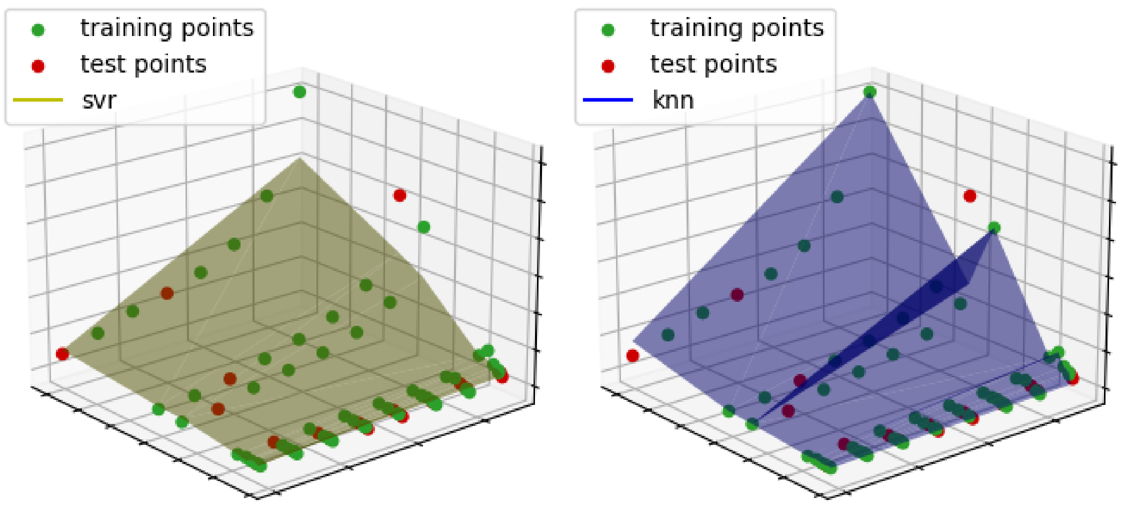


Fig. 9: Regression surfaces for *images_merger* module.

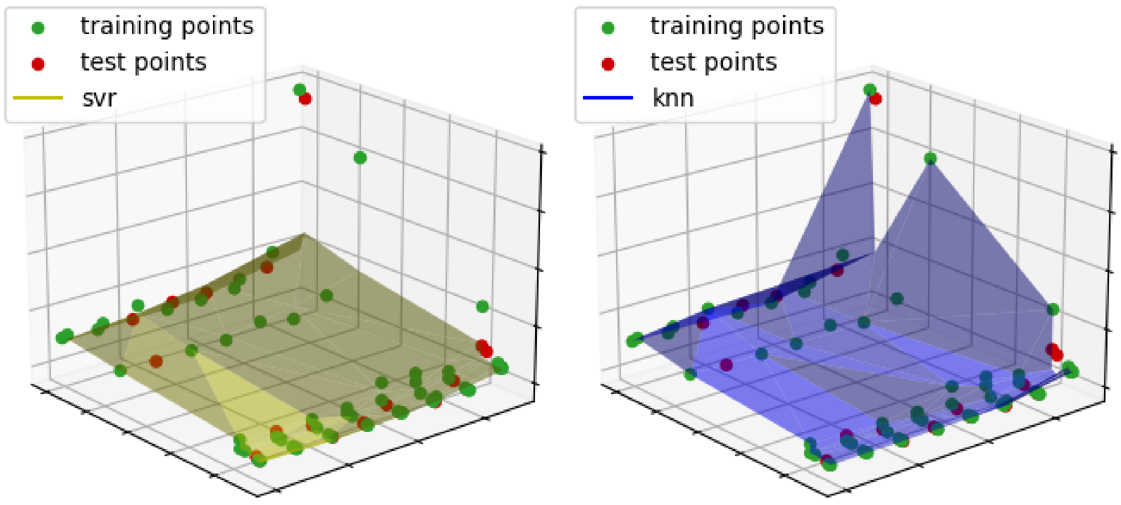


Fig. 10: Regression surfaces for *video_splitter* module.

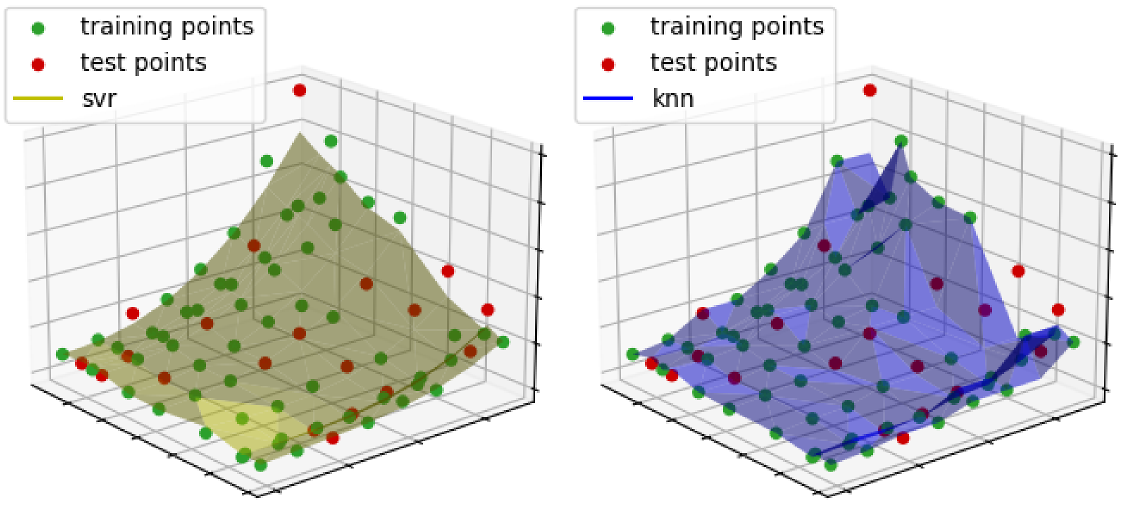


Fig. 11: Regression surfaces for *face_recogniser* module.

[11] Vladimir Vapnik, *Support-Vector Networks by Corinna Cortes*, FILL IT, 1995.

[12] Wikipedia, *Radial basis function kernel*, Wikipedia, 03.03.2021.

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

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

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

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

[16] wikipedia.org, *Minkowski distance*, wikipedia.org, visited 10.04.2021.

[17] wikipedia.org, *Euclidean distance*, wikipedia.org, visited 10.04.2021.

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

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

[20] "The history of algorithmic complexity," .