Self Organizing Maps for Algorithm Classification

Rares Folea
Department of Computer Science
Politehnic University of Bucharest
Bucharest, Romania
rares.folea@stud.acs.upb.ro

Abstract

This paper presents the usage of a self-organizing map, as an unsupervised neural network that reduces the input dimensional in order to represent the distribution of algorithms embeddings as a map. The used algorithms embeddings are based on r-Complexity [2], a slightly different complexity asymptotic notation than the traditional Bachmann-Landau complexity notation.

1 Introduction

Being aware of the **meaning** of **source code** representations is, undoubtedly, a *profoundly* researched topic in Computer Science. While there are many fields of applications, such as computerized *discoveries of bugs*, *detecting encryption functions in malware*, and *automated solutions for code generation*, **automatic code labelling** is a topic that we consider worth discussing, because it is a problem that requires a **subtle understanding** on how code behaves. It can reveal solutions for many more additional problems. The problem that we approach has an *inherent degree of burdensome* as this requires not only profound judgment calls on the semantics of the code but also the capability of clustering algorithms based on their **commitment** and **utility**, aspects that may not be obvious only from a perfunctory analysis. We will study in this research computer programs written in **C++**, a language that has expanded significantly over time, with modern programming language now having object-oriented, generic, and functional features. Nevertheless, C++ provides, in addition, low-level facilities such as plenty solutions for bare memory manipulation. C++ is a common choice for competitive programming challenges, as there have been developed powerful *compilers* and *optimizers* that are capable of obtaining the most out of an efficient implementation from a coding idea, providing best runtime results in class.

The self-organized map is an architecture suggested for artificial neural networks that has the property of effectively creating spatially organized internal representations of various features of input signals and their abstractions [5]. To understand how the training evolves we are going to plot the quantization and topographic error of the self-organizing map at each step. This is particularly important to estimate the number of iterations to run. To have an overview of how the samples are distributed across the map a scatter chart can be used where each dot represents the coordinates of the winning neuron.

2 Self-organizing maps

Learning in the self-organizing map aims to **make various regions of the network respond to certain input patterns in a similar way**. This is largely *inspired simply because of the way visual, auditory, and other sensory information is interpreted in different segments of the human brain's cerebral cortex* [8], [3].

2021-2022: Self-Organizing System project submission. AI master @ ACS, UPB, Bucharest, Romania

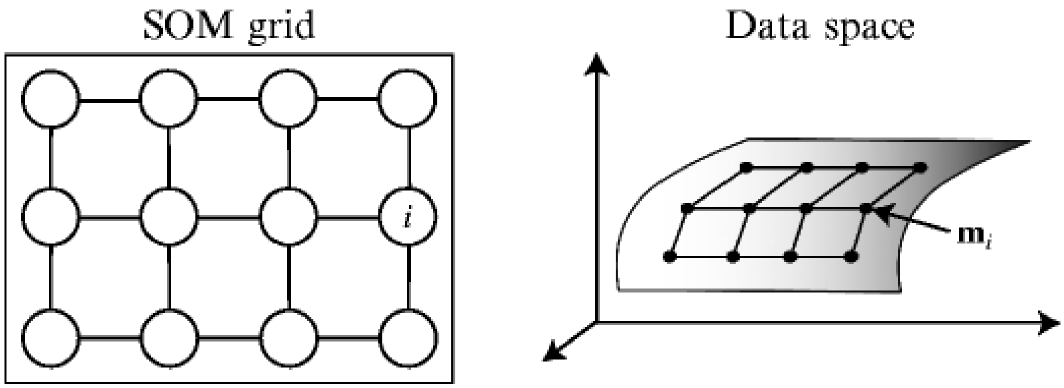

Figure 1: The mapping between the self-organizing map grid and the data space.

SOMs were first introduced in the 1980s by Teuvo Kohonen [5]. A SOM is a sort of neural network that handles the training procedure via competitive learning rather than the error-correction learning methods utilized by other neural networks (e.g., back-propagation with gradient descent).

SOMs have been successfully used in multiple areas of research, such as project prioritization and selection, seismic facies analysis for oil and gas exploration, failure mode and effects analysis and creation of artwork.

2.1 The quality of feature map

Quantization error and topographical error are main measurements to assess the quality of SOM. Quantization error is the average difference of the input samples compared to its corresponding winning neurons (BMU). It assesses the accuracy of the represented data, therefore, it is better when the value is smaller [6].

Topographical error assesses the topology preservation. It indicates the number of the data samples having the first best matching unit (BMU1) and the second best matching unit (BMU2) being not adjacent. Therefore, the smaller value is better [4].

Using these two metrics, we will evaluate the quality of the resulted feature map after the learning process for the self-organizing map for the different test scenarios that we have prepared.

To understand how the training evolves we can plot the quantization and topographic error of the SOM at each learning step. This is important when estimating the number of iterations to run in the SOM training.

3 Algorithm2Embedding

As shown in [2] r-Complexity is a refined complexity calculus model that provides a new asymptotic notation that offers better complexity feedback for similar programs than the traditional Bachmann-Landau notation, providing subtle insights even for algorithms that are part of the same conventional complexity class.

The following notations and names will be used for describing the asymptotic behavior of a algorithm's complexity characterized by a function, $f : \mathbb{N} \longrightarrow \mathbb{R}$.
We define the set of all complexity calculus $\mathcal{F} = \{f : \mathbb{N} \longrightarrow \mathbb{R}\}$
Assume that $n, n_0 \in \mathbb{N}$. Also, we will consider an arbitrary complexity function $g \in \mathcal{F}$. Acknowledge the following notations $\forall r \neq 0$:

Definition 3.1 ***Big r-Theta****: This set defines the group of mathematical functions similar in magnitude with $g(n)$ in the study of asymptotic behavior. A set-based description of this group can be expressed*

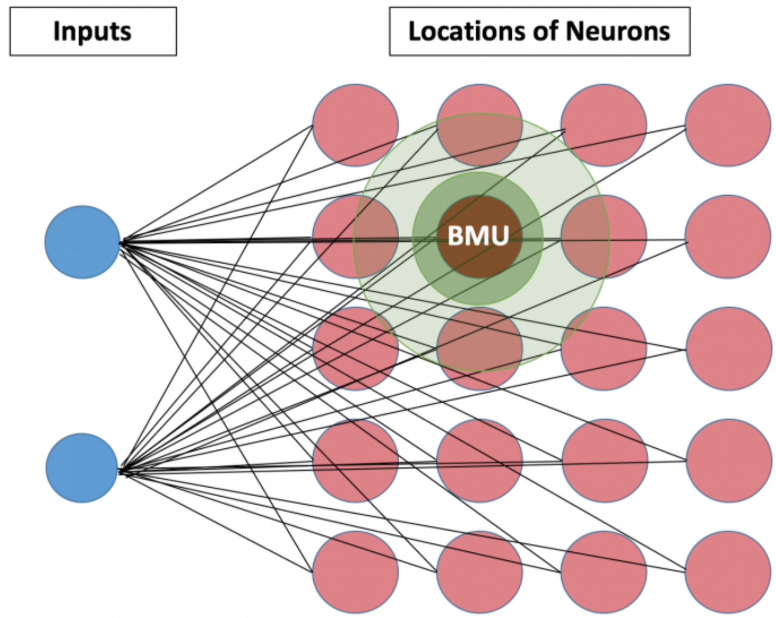

Figure 2: The process of choosing the BMU for arbitrary inputs in a self-organizing map.

as:

$$\Theta_r(g(n)) = \{f \in \mathcal{F} \mid \forall c_1, c_2 \in \mathbb{R}_+^* \ s.t. c_1 < r < c_2, \exists n_0 \in \mathbb{N}^* \\ s.t. \ c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n) \ , \ \forall n \geq n_0\}$$

The big **r-Theta** notation proves to be useful also in creating Dynamic code embeddings. The idea behind these embeddings is simple: try to automatically provide estimations, for various metrics, the r-Theta class for the analyzed algorithm (usually with unknown Bachmann–Landau Complexity).

A generic solution to providing automatic estimation of r-Complexity is [1]:

$$f(n) = \sum_{t=1}^{y} \sum_{k=1}^{x} c_k \cdot n^{p_k} \cdot log_{l_k}^{j_k}(n) \cdot e_t^n \cdot$$

Yet, we will need to instantiate the model thereby presented to analyze tangible metrics with respect to the input size.

In this research, we will attempt to fit a simplified version of the generic expression as a Big r-Theta function, described generic by one of the following function:

$$\begin{cases} r \cdot log_2^p log_2(n) + X \\ r \cdot log_2^p(n) + X \\ r \cdot p^n + X, p < 1 \\ r \cdot n^p + X \\ r \cdot p^n + X, p > 1 \\ r \cdot \Gamma(n) + X \end{cases}.$$

We analyze an algorithm by the associated r-embedding. It can be built on top of any metrics collected. Using the `perf` profiler, we obtained and used the following metrics:

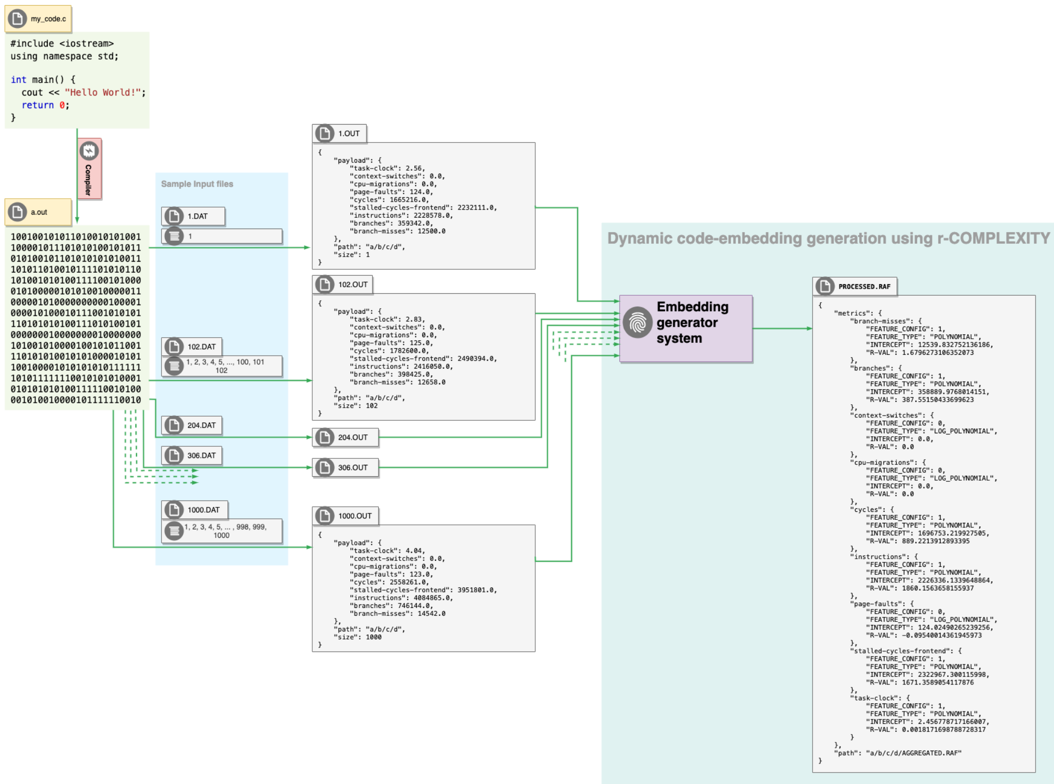

Figure 3: The process of generating a code-embedding based on r-Complexity. Based on the metrics collected by the profiler, we compute the dynamic-code embeddings.

- branch-misses
- branches
- context-switches
- cpu-migrations
- cycles
- instructions
- page-faults
- stalled-cycles-frontend
- task-clock

Therefore, based on these metrics, our code can be processed as a 36-long dynamic-code embedding[1], built using approximations of r-Theta complexity, containing:

- `branch-misses_FEATURE_CONFIG`
- `branch-misses_FEATURE_TYPE`
- `branch-misses_INTERCEPT`
- `branch-misses_R-VAL`
- `branches_FEATURE_CONFIG`
- `branches_FEATURE_TYPE`
- `branches_INTERCEPT`
- `branches_R-VAL`
- `context-switches_FEATURE_CONFIG`
- `context-switches_FEATURE_TYPE`
- `context-switches_INTERCEPT`
- `context-switches_R-VAL`
- `cpu-migrations_FEATURE_CONFIG`
- `cpu-migrations_FEATURE_TYPE`

[1]We also provide a solution for generating a different set of embeddings, based on the time utility under Linux. Yet, these embeddings provide better results for the dataset that we have analyzed.

cpu-migrations_INTERCEPT
cpu-migrations_R-VAL
cycles_FEATURE_CONFIG
cycles_FEATURE_TYPE
cycles_INTERCEPT
cycles_R-VAL
instructions_FEATURE_CONFIG
instructions_FEATURE_TYPE
instructions_INTERCEPT
instructions_R-VAL
page-faults_FEATURE_CONFIG
page-faults_FEATURE_TYPE
page-faults_INTERCEPT
page-faults_R-VAL
stalled-cycles-frontend_FEATURE_CONFIG
stalled-cycles-frontend_FEATURE_TYPE
stalled-cycles-frontend_INTERCEPT
stalled-cycles-frontend_R-VAL
task-clock_FEATURE_CONFIG
task-clock_FEATURE_TYPE
task-clock_INTERCEPT
task-clock_R-VAL

4 Dataset

We have prepared two datasets:

- *math_dataset.csv* containing 5949 code-embeddings, out of which 4937 for problems that are not related to math and 1012 for problems that are not classified as math-related.
- *small_math_dataset.csv*, a subset of *math_dataset.csv*, containing 200 code-embeddings, out of which 100 for problems that are not related to math and 100 for problems that are not classified as math-related.

A sample dataset entry, containing 2 code-embeddings, 1 for a problem that is not related to math and 1 for a problem that is not classified as math-related:

```
[CSV columns names]

branch-misses_FEATURE_CONFIG,branch-misses_FEATURE_TYPE,
branch-misses_INTERCEPT,branch-misses_R-VAL,
branches_FEATURE_CONFIG,branches_FEATURE_TYPE,
branches_INTERCEPT,branches_R-VAL,
context-switches_FEATURE_CONFIG,context-switches_FEATURE_TYPE,
context-switches_INTERCEPT,context-switches_R-VAL,
cpu-migrations_FEATURE_CONFIG,cpu-migrations_FEATURE_TYPE,
cpu-migrations_INTERCEPT,cpu-migrations_R-VAL,
cycles_FEATURE_CONFIG,cycles_FEATURE_TYPE,
cycles_INTERCEPT,cycles_R-VAL,
instructions_FEATURE_CONFIG,instructions_FEATURE_TYPE,
instructions_INTERCEPT,instructions_R-VAL,
page-faults_FEATURE_CONFIG,page-faults_FEATURE_TYPE,
page-faults_INTERCEPT,page-faults_R-VAL,
stalled-cycles-frontend_FEATURE_CONFIG,stalled-cycles-frontend_FEATURE_TYPE,
stalled-cycles-frontend_INTERCEPT,stalled-cycles-frontend_R-VAL,
task-clock_FEATURE_CONFIG,task-clock_FEATURE_TYPE,
task-clock_INTERCEPT,task-clock_R-VAL,
label

[entry 1]
0,LOG_POLYNOMIAL,12176.278639860398,5.274471559958294,
```

```
1,POLYNOMIAL,354489.8418811041,31.023435300379777,
0,LOG_POLYNOMIAL,0.0,0.0,
0,LOG_POLYNOMIAL,0.0,0.0,
3,LOG_POLYNOMIAL,1657272.38324269,1.4722602961244892e-05,
1,POLYNOMIAL,2196944.89849295,137.26687233920643,
0.9,FRACTIONAL_POWER,119.36451460061674,0.7558436224332522,
5,LOG_POLYNOMIAL,2277557.4855760685,1.994809269682738e-11,
3,POLYNOMIAL,2.470760207032896,-2.172036539848529e-10,
[label 1]
0

[...]

[entry 200]
1,POLYNOMIAL,12370.712368796743,0.9930555401849003,
1,POLYNOMIAL,355267.67041624436,818.7959073312177,
0,LOG_POLYNOMIAL,0.0,0.0,
0,LOG_POLYNOMIAL,0.0,0.0,
1,POLYNOMIAL,1673084.5351177657,1735.1433240315066,
1,POLYNOMIAL,2201813.5172021347,3798.310753165602,
0,LOG_POLYNOMIAL,118.69266672203334,0.06013108223346909,
1,POLYNOMIAL,2315580.396807521,1911.1634798457628,
1,LOG_POLYNOMIAL,2.3873598897146207,0.00025835398225644443,
[label 200]
1
```

5 Experiments

We used **MiniSom** [7] as the starting framework for testing the performance of **Self Organizing Maps** on these datasets. MiniSom is a minimalistic and numpy based implementation of the Self Organizing Maps, able to convert nonlinear statistical relationships between high-dimensional data items into simple geometric relationships on a **low-dimensional display**.

5.1 Small dataset

5.1.1 Optimal Neighborhood function

We have tried 100k iteration training for various Neighborhood function, using the entire feature set available for this reduced dataset:

- gaussian
- mexican-hat
- bubble
- triangle

The most relevant results were obtained when using the gaussian function, as presented in the below graphs. This has outputted the lowest quantization error and the best spread of dataset entries across multiple neurons.

5.1.2 Optimal Activation Distance Function

We have tried 100k iteration training for various Activation Distance, using the entire feature set available for this reduced dataset:

- euclidean

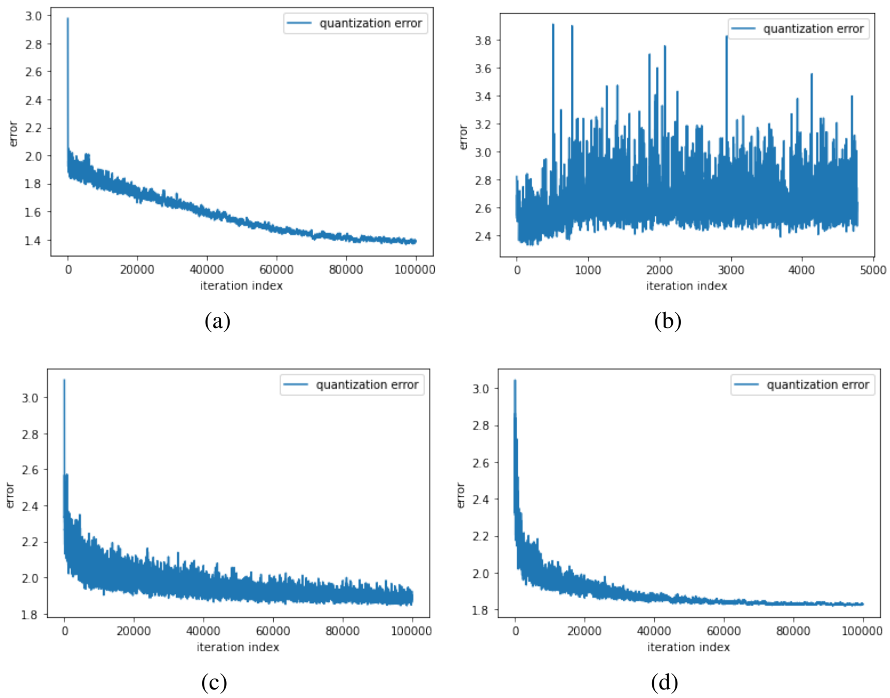

Figure 4: The Quantization error obtained when using the Neighborhood function: (a) **gaussian** (b) **mexican-hat** (c) **bubble** (d) **triangle**, during a 100k iteration training of a 6x6 SOM.

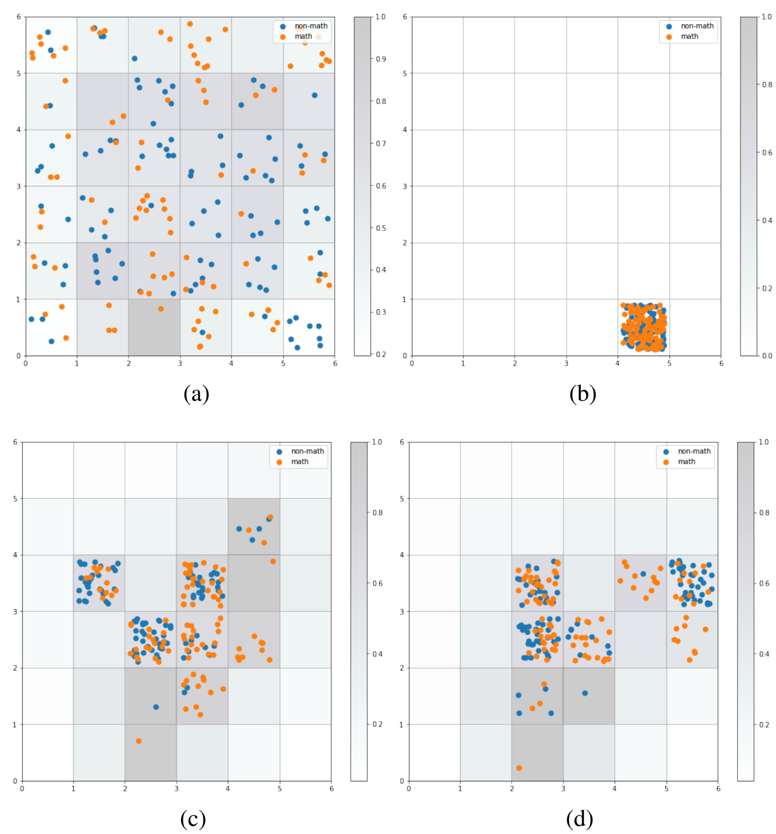

Figure 5: The distribution across the self-organizing map of the dataset obtained when using the Neighborhood function: (a) **gaussian** (b) **mexican-hat** (c) **bubble** (d) **triangle**, during a 100k iteration training of a 6x6 SOM.

- cosine
- manhattan
- chebyshev

The most relevant results were obtained when using the **euclidean** and **cosine** function, as presented in the below graphs. These two activation functions has outputted the lowest quantization error and the best spread of dataset entries across multiple neurons.

5.1.3 Feature selection

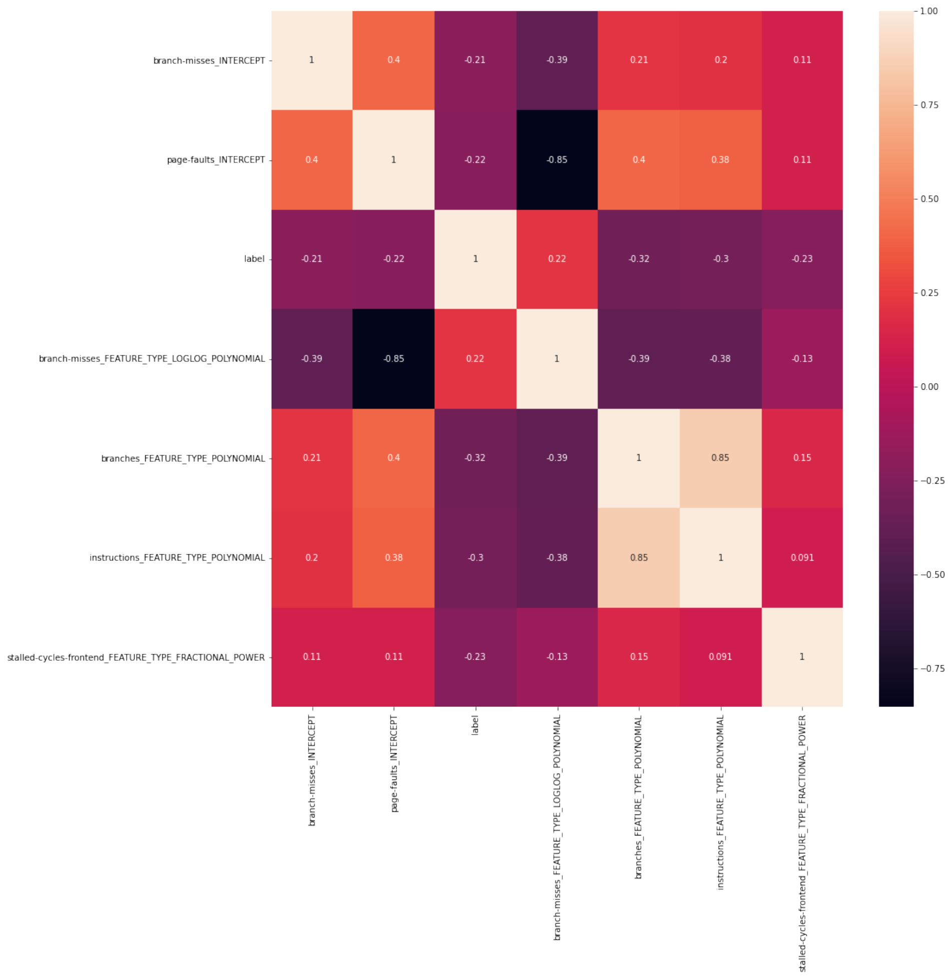

Figure 8: Heatmap corresponding to the top features that have a Pearson correlation (in absolute value) with the label at least **.2**.

We ran 1M training-iteration using the Gaussian Neighbourhood function on the small dataset, containing:

- All features provided in the dataset.
- Some features provided in the dataset, chosen based on correlation factors. To begin with, we investigated the correlations between the features and the feature of the targeted algorithm in the dataset. We have used the pandas implementation of the Pearson correlation.
  The selected features were:
  - branch-misses_INTERCEPT,

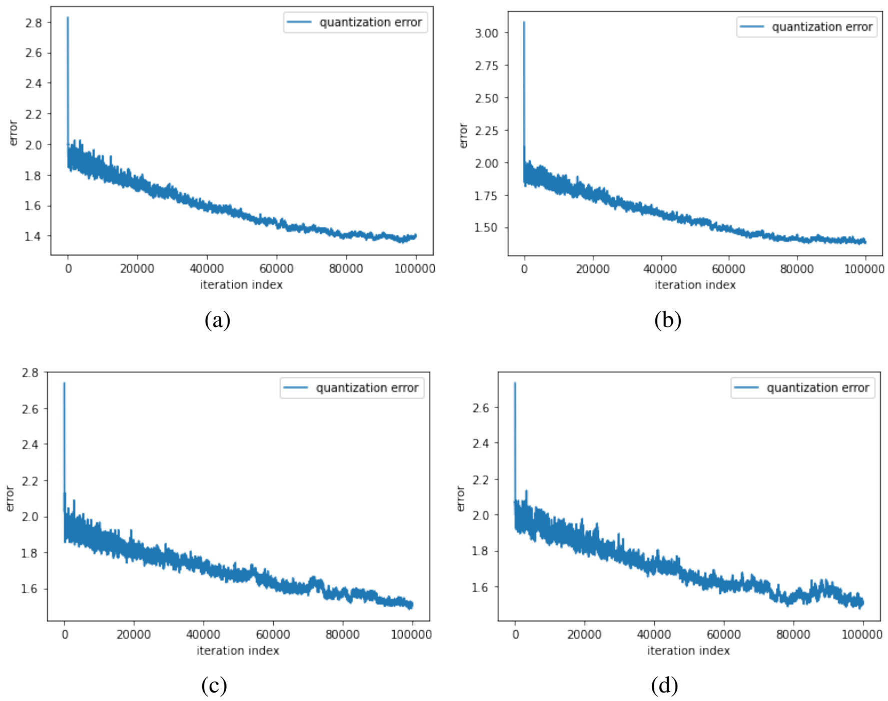

Figure 6: The Quantization error obtained when using the Activation function: (a) **euclidean** (b) **cosine** (c) **manhattan** (d) **chebyshev**, during a 100k iteration training of a 6x6 SOM.

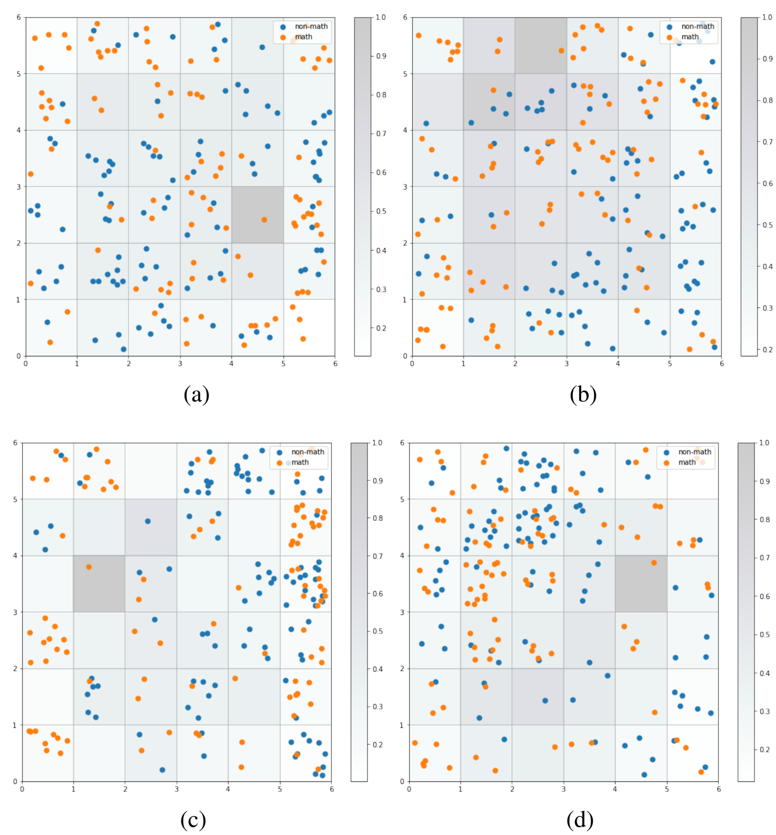

Figure 7: The distribution across the self-organizing map of the dataset obtained when using the Activation function: (a) **euclidean** (b) **cosine** (c) **manhattan** (d) **chebyshev**, during a 100k iteration training of a 6x6 SOM.

– page-faults_INTERCEPT,

– branch-misses_FEATURE_TYPE_LOGLOG_POLYNOMIAL,

– branches_FEATURE_TYPE_POLYNOMIAL,

– instructions_FEATURE_TYPE_POLYNOMIAL,

– stalled-cycles-frontend_FEATURE_TYPE_FRACTIONAL_POWER.

We have summarized the results for both scenarios in the following subsubsections.

5.1.4 Using all dataset features

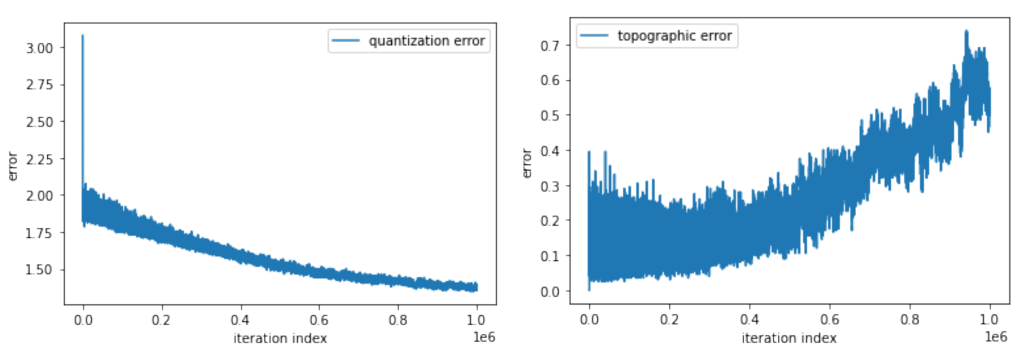

Figure 9: [Small dataset - All features] To understand how the training evolves we can plot the quantization and topographic error of the SOM at each step.

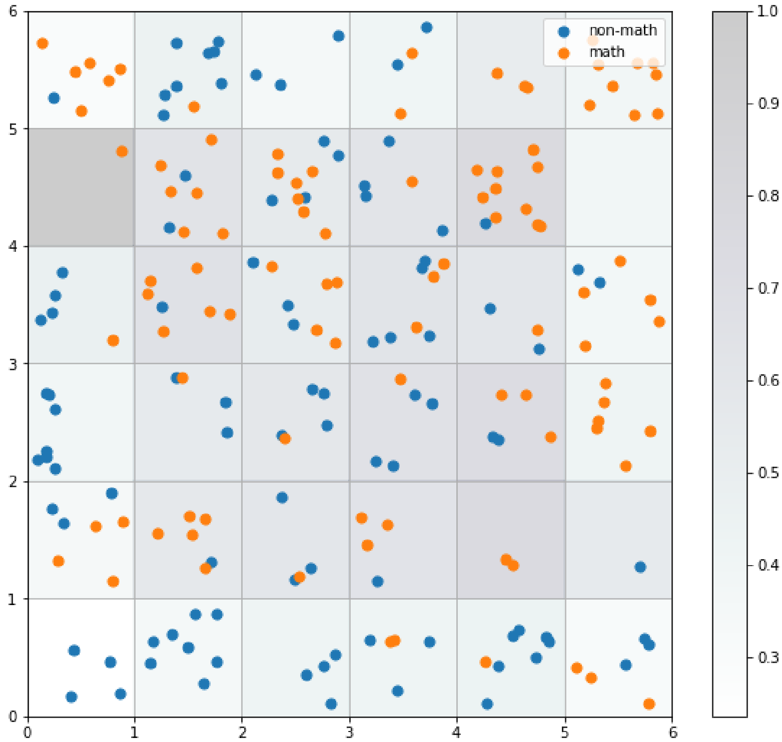

Figure 10: [Small dataset - All features] To visualize the result of the training we can plot the distance map (U-Matrix) using a pseudocolor where the neurons of the maps are displayed as an array of cells and the color represents the (weights) distance from the neighbour neurons. To have an overview of how the samples are distributed across the map a scatter chart can be used where each dot represents the coordinates of the winning neuron.

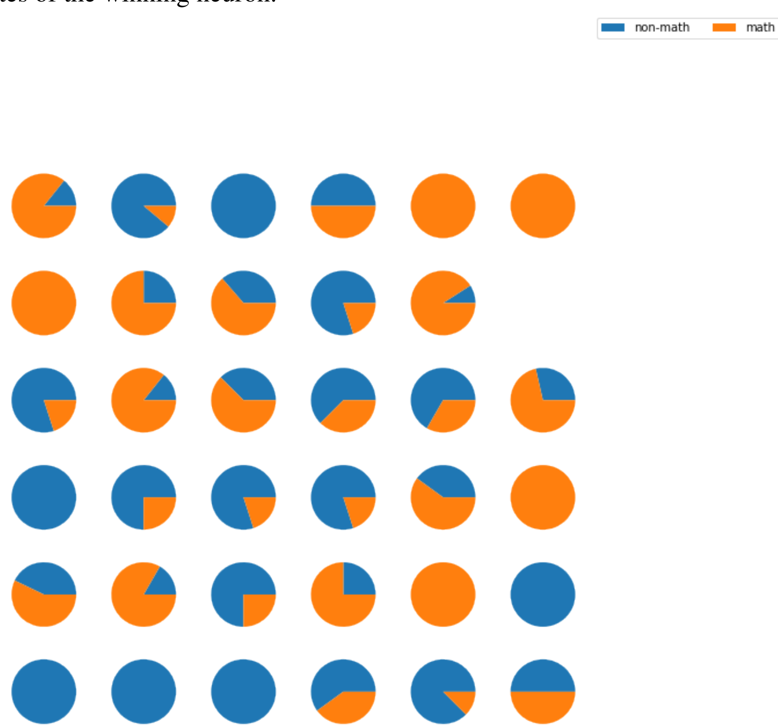

Figure 11: [Small dataset - All features] We can visualize the proportion of samples per class falling in a specific neuron using this pie chart per neuron.

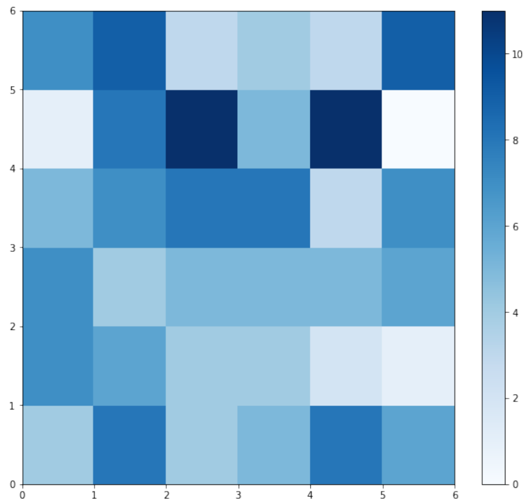

Figure 12: [Small dataset - All features] To have an idea of which neurons of the map are activated more often we can create another pseudocolor plot that reflects the activation frequencies.

5.1.5 Using some dataset features

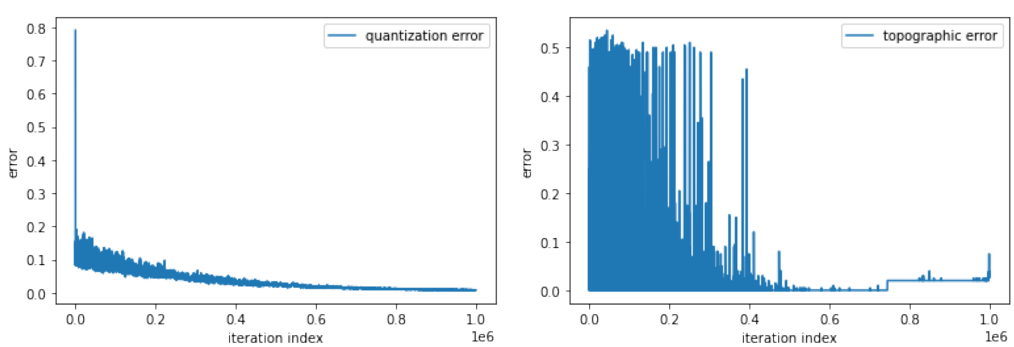

Figure 13: [Small dataset - Some features] To understand how the training evolves we can plot the quantization and topographic error of the SOM at each step.

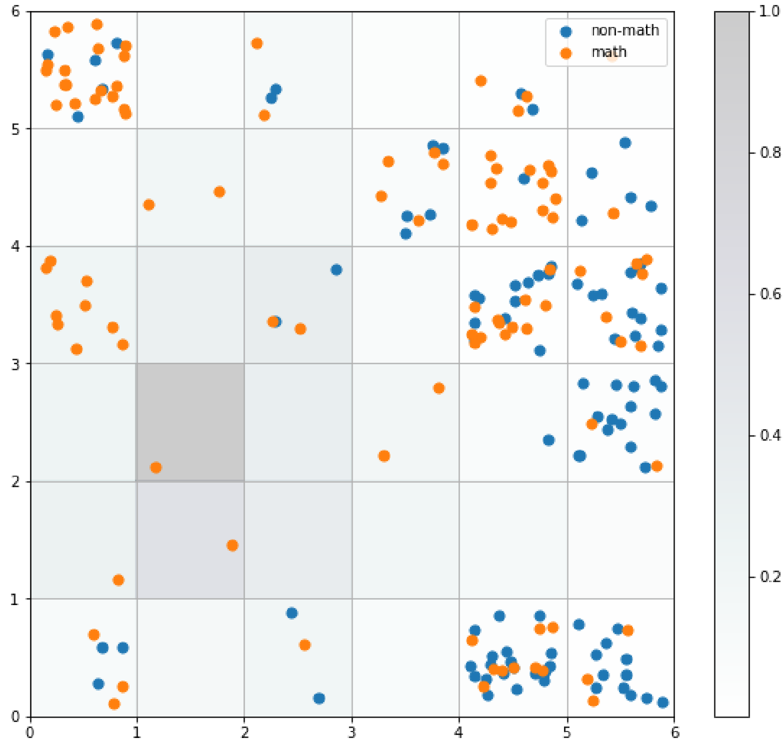

Figure 14: [Small dataset - Some features] To visualize the result of the training we can plot the distance map using a pseudocolor where the neurons of the maps are displayed as an array of cells and the color represents the (weights) distance from the neighbour neurons. To have an overview of how the samples are distributed across the map a scatter chart can be used where each dot represents the coordinates of the winning neuron.

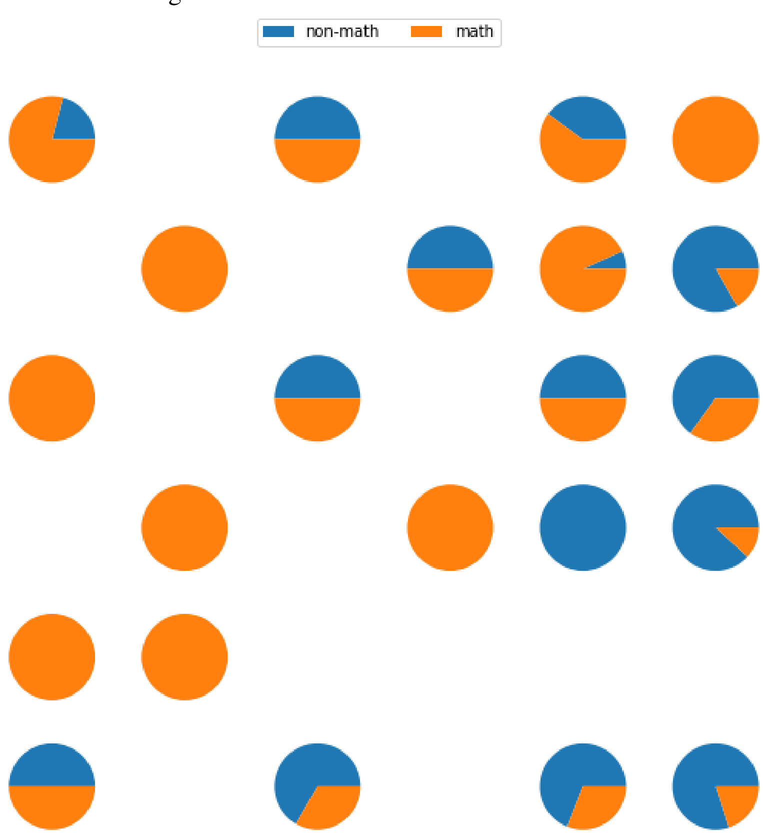

Figure 15: [Small dataset - Some features] We can visualize the proportion of samples per class falling in a specific neuron using this pie chart per neuron.

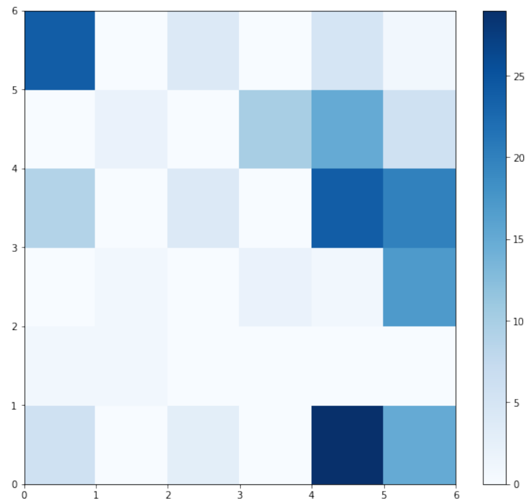

Figure 16: [Small dataset - Some features] To have an idea of which neurons of the map are activated more often we can create another pseudocolor plot that reflects the activation frequencies.

5.2 Large dataset

For this experiment, using the large, containing over 5000+ entries, we have used only a subset of the dataset features with high correlation, gaussian neighborhood function and cosine function as an activation in the SOM model. We trained a 216x216 SOM network for 10M steps.

The results are available open-source, at: `https://github.com/raresraf/AlgoRAF/blob/master/prototype/SelfOrganizingMap/allDataset-SOM-SomeFeatures.ipynb`

The large 216x216 SOM was able to perform decent in separating the two types of problems based on the labels.

6 Conclusion

When dealing with a supervised classification problem, such as algorithm classification, SOMs can be trained and later used to obtain relevant models, as well as helping visualize the data in a lower dimensional space.

Sometimes, reducing the **Quantization error** has the side effect of increasing the **Topographical error**, which indicates samples having the first best matching unit (BMU1) and the second best matching unit (BMU2) being not adjacent.

For the small dataset, using only a subset of the dataset features with high correlation increased the performance of the SOM model, when used with the gaussian neighborhood function and cosine function as an activation.

For the experiment on the large dataset, we trained a 216x216 SOM network for 10M steps. It took 1.5 days to train on a modern i7 8-core CPU. The results have been satisfactory, in which we saw a visual split between the two main labels.

References

[1] Alexandru Calotoiu. *Automatic Empirical Performance Modeling of Parallel Programs*. PhD thesis, Technische Universität, 2018.

[2] Rares Folea and Emil-Ioan Slusanschi. A new metric for evaluating the performance and complexity of computer programs: A new approach to the traditional ways of measuring the

Figure 17: A visual representation of a 216x216 SOM network trained for 10M steps on the large dataset. The large scale images for the visual representation of the SOM is available open-source, at: `https://raw.githubusercontent.com/raresraf/AlgoRAF/master/prototype/SelfOrganizingMap/som_seed.png`

Figure 18: A visual representation of the decisions of neurons in the 216x216 SOM network trained for 10M steps on the large dataset. The large scale images for the visual representation of the SOM is available open-source, at: `https://raw.githubusercontent.com/raresraf/AlgoRAF/master/prototype/SelfOrganizingMap/som_seed_pies.png`

complexity of algorithms and estimating running times. In *2021 23rd International Conference on Control Systems and Computer Science (CSCS)*, pages 157–164. IEEE, 2021.

[3] Simon Haykin. Self-organizing maps. *Neural networks-A comprehensive foundation, 2nd edition, Prentice-Hall*, 1999.

[4] Kimmo Kiviluoto. Topology preservation in self-organizing maps. In *Proceedings of International Conference on Neural Networks (ICNN'96)*, volume 1, pages 294–299. IEEE, 1996.

[5] Teuvo Kohonen. The self-organizing map. *Proceedings of the IEEE*, 78(9):1464–1480, 1990.

[6] Teuvo Kohonen. Self-organizing maps: ophmization approaches. In *Artificial neural networks*, pages 981–990. Elsevier, 1991.

[7] Giuseppe Vettigli. Minisom: minimalistic and numpy-based implementation of the self organizing map, 2018.

[8] Chr Von der Malsburg. Self-organization of orientation sensitive cells in the striate cortex. *Kybernetik*, 14(2):85–100, 1973.