Perlin Noise-Based Genetic Algorithm Classifier

Grigorescu Andrei-Viorel

Abstract

Procedural noise, and particularly Perlin noise, which allows for generation of continuous pseudo-random values, has previously seen use in Artificial Intelligence as a mean to either assist in random choice making, or for augmentation of datasets with a limited size [1]. While there are experiments where Perlin noise was used to initialize the default weights in a convolutional neural network [2], or where parameter optimization was used to determine optimal Perlin noise parameters for the purpose of generating adversarial input for object detection networks [3], the use of Perlin noise's properties has been rather limited in this domain to mostly data augmentation. Worth mentioning are recent advances in diffusion models which seek to generate data starting from random noise by learning how to reverse the process of adding noise to the data [4]. This paper intends to continue the idea of improving the parameters of a Perlin noise so that it can be used to solve a classification problem. If successful, a set of parameters for a Perlin noise that were evolved using a genetic algorithm will encode a non-linear solution for classifying entries with an arbitrary number of features in such a way that it will be quick to evaluate, especially given the HPC implementations inspired by Perlin noise's use in computer graphics, and which will be able to exploit the computer graphics techniques for using Perlin noise, such as layering and other manipulations [5].

ACM CCS Classification

•Computing methodologies → Machine learning → Machine learning approaches → Classification and regression trees

AMS Classification

68T01 - General topics in artificial intelligence

Keywords

Artificial intelligence; Machine learning; Classification; Procedural noise; Genetic algorithm.

Table of contents

Abstract 1
ACM CCS Classification 1
AMS Classification 1
Keywords 1
Table of contents 2
Introduction 3
Artificial intelligence (AI) and Machine Learning (ML) 3
Perceptron: XOR problem 3
Support vector machines (SVM), artificial neural networks (ANN) 3
Procedural noise: Perlin noise 3
Current applications of Perlin noise in AI and ML 3
Genetic algorithm classifier based on Perlin noise 3
Natural description and intuitive explanation 3
Formal model of the classifier 5
Experimental analysis 7
Research hypothesis 7
Research questions 7
Experimental procedure 7
Original approach 7
Description of chosen experiments 7
Study case 8
Experiment 1: Logic gates 8
Experiment 2: Iris flower dataset 11
Related work 29
Genetic algorithms 29
Support vector machines 29
Artificial neural networks 29
Conclusion 30
Further work 30
Bibliography 31
Table of figures 32

Introduction

Artificial intelligence (AI) and Machine Learning (ML)

Perceptron: XOR problem

Support vector machines (SVM), artificial neural networks (ANN)

Procedural noise: Perlin noise

Current applications of Perlin noise in AI and ML

Genetic algorithm classifier based on Perlin noise

Natural description and intuitive explanation

In the following, the train of thought that led to the idea of the proposed classifier will be explained, using visual parallels to the transformations discussed.
Upon studying of the hill climbing algorithm, a simple search-based solution solving algorithm, it is learnt that each problem is associated a *problem landscape*.
A parallel was drawn between this concept and that of terrain generation using procedural noise for virtual worlds.
The question that arose from this association of concepts was the following: can a procedural noise come to model, via means of artificial intelligence and machine learning, the problem landscape with enough precision to provide an easily evaluable function for the fitness of said problem's solutions?

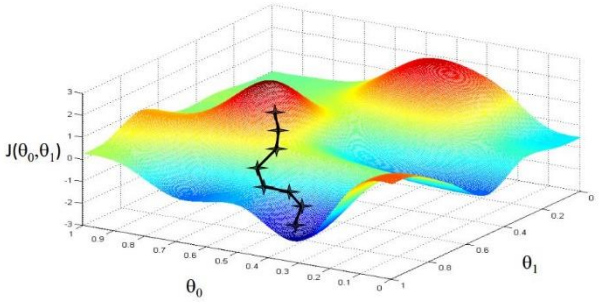

Figure 1 An example of loss values in a solution space (Source: https://umu.to/blog/2018/06/29/hill-climbing-irl)

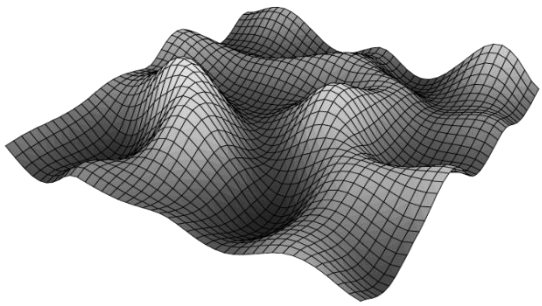

Figure 2 An example of terrain mesh generated using Perlin noise (Source: https://www.scratchapixel.com/lessons/procedural-generation-virtual-worlds/perlin-noise-part-2/perlin-noise-terrain-mesh)

The proposed classifier uses the fact that procedural noises generate pseudo-random values; that is to say: given the same parameters, including the pseudo-random number generator seed, the same results can be achieved on each iteration.
To enable flexibility of the procedural noise generated terrain mesh, several transformations can be applied. This not only provides more options for generating a landscape, but also offers mutations that are appropriate for use in genetic algorithms, such as soft or hard mutations.
One example of transformation is that of using scaling and translation offsets across each input dimension, which in image manipulation, would be analogous to resizing and panning a picture.

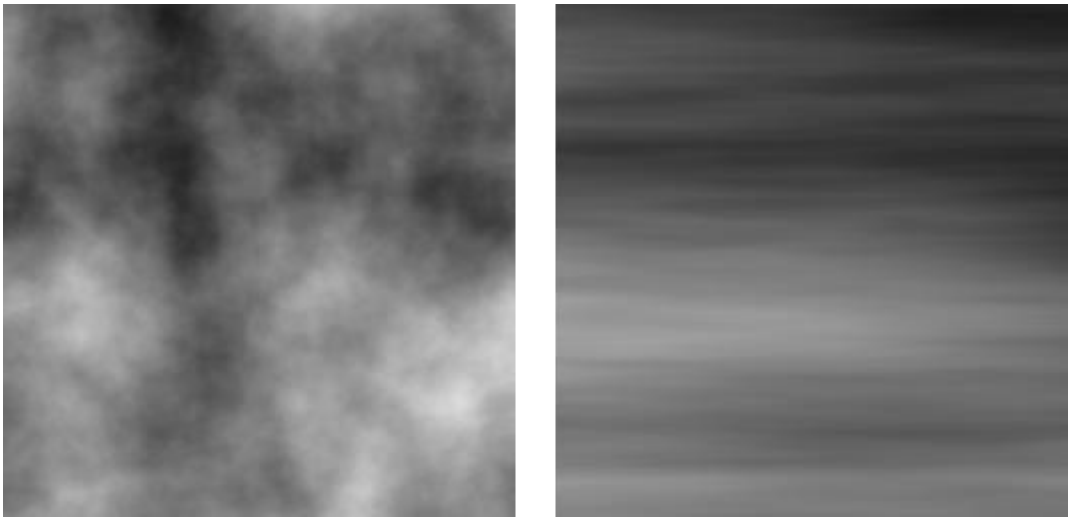

Figure 3 Example of scaling across a dimension in a Perlin noise

The number of octaves, or the turbulence of the Perlin noise, can also be manipulated to adjust the granularity of the details in the mesh.

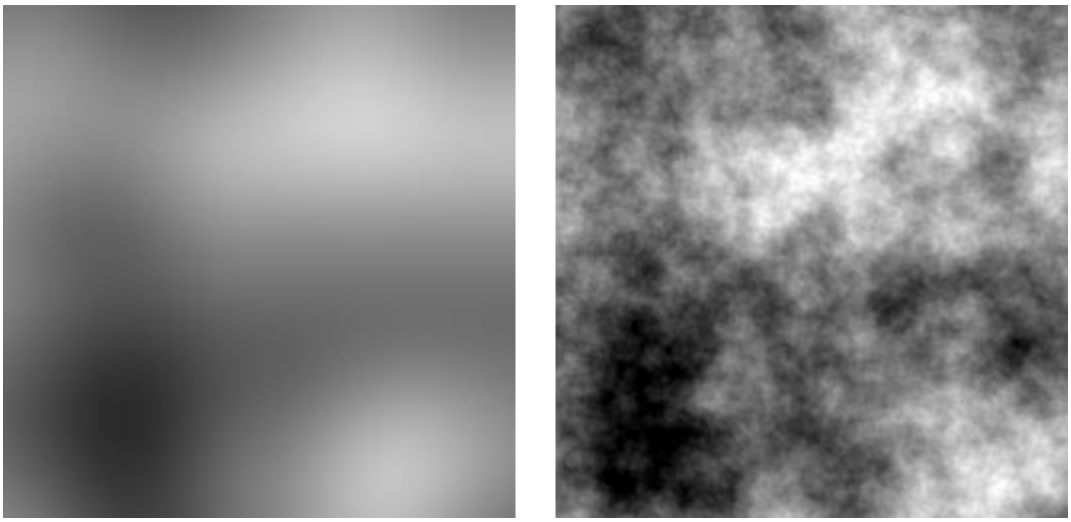

Figure 4 Changing the number of octaves or manipulating turbulence in a Perlin noise

Finally, to obtain a binary result, a threshold is applied to the values generated by the procedural noise, threshold which can also be manipulated to produce a range of similar results. This operation, in context of image editing, would be comparable to applying maximum contrast and varying brightness to the generated noise.

Figure 5 Manipulating the threshold applied to the same Perlin noise by changing brightness value

Moreover, techniques derived from the use of procedural noise in computer graphics can be applied to obtain more means of manipulating the noise, such as generating procedural noises with a given number of supplementary dimensions, then getting a slice in a smaller number of dimensions across one or more free parameters. These parameters can now be used to obtain variations which are similar, but different in a manner unlike scaling or translating across the bound dimensions.

Formal model of the classifier

Definition of the chromosome

The classifier shown here uses Perlin noise as the procedural noise function and is described by a vector of parameters which define the Perlin noise instance, the transformations applied to the input values, unbound variables and the threshold that is applied in the end. In the context of genetic algorithms that are used to search for an appropriate classifier, this will be further called *chromosome*. For an input with *N* features, a Perlin noise with *N+1* dimensions will be considered, and the chromosome $c \in C$, with $C = \mathbb{N}^2 \times \mathbb{R}^+ \times \mathbb{R}^2$ will have the following structure:

$$c = [s, o, t, s_1, o_1, \cdots, s_N, o_N, z],$$

where the symbols have the following meanings:

- $s \in \mathbb{N}$, seed for the pseudo-random number generator used in the Perlin noise
- $o \in \mathbb{N}$, octaves in the Perlin noise, also called turbulence
- $t \in \mathbb{R}^+$, threshold value
- $s_i \in \mathbb{R}$, scale across *i*-th dimension, $i = \overline{1,\ N}$
- $o_i \in \mathbb{R}$, offset or translation across *i*-th dimension, $i = \overline{1,\ N}$
- $z \in \mathbb{R}$, offset across *(n+1)*-th dimension

Of course, given different transformations to be applied to the Perlin noise, the structure of the chromosome is subject to change.

For the initial population of the genetic algorithm, the values for the parameters will be selected from uniform distributions within the ranges described. While concrete numbers are given in the following part, those can and should be subjected to hyper-parametrization so that they are tailored to the problem at hand.

- $s = \overline{0,100000000}$

- $o = \overline{1, 256}$
- $t \in [0.1, 0.9]$
- $s_i \in [10, 10]$
- $o_i \in [10, 10]$
- $z \in [10, 10]$

Description of the genetic algorithm, evolution tactic and fitness function

A genetic algorithm will be used to search for an optimal solution that will binarily classify inputs, with multiclass problems being solved in a one-versus-all fashion. In order to be able to describe a solution as optimal, a fitness function *f* is used in order to compare multiple solutions.

$$f: C \to \mathbb{R}$$

A fitness function will primarily weigh proportionally with the accuracy of the classifier, although other components can be added to guide the search. For example, it could be desirable for the fitness function to be inversely proportional with *o*, the turbulence of the Perlin noise within the chromosome, or to similarly use high scale factors so that simple models are obtained.
Finally, an optimal solution $c_{optimal}$ is one which maximum fitness across populations in every generation.

$$c_{optimal} = \max_{f(c)} c_{i,gen}, i = \overline{1, popSize}, gen = \overline{1, noGen},$$

where *popSize* represents the size of a chromosome population and *noGen* represents the number of generations, or iterations, for which to run the genetic algorithm.
The evolution tactic that will be used is that of steady state evolution. At each iteration, each one of two parents are selected by taking the most fit individual out of two randomly chosen individuals in the population. Then, an offspring is obtained by applying the crossover operator on the two parents, followed by a random soft mutation. If the obtained offspring is better than the least fit individual in the population, that individual will be replaced by the new one. This kind of evolution tactic results in a monotonous evolution of both the average and maximum fitness in the population over the generations, keeping the solutions with traits that lead to a better fitness than existing ones, at an increased risk of getting trapped in a local optimum.

Definition of soft mutation operator

The soft mutation operator takes one chromosome, c_{old}, and returns a new chromosome, c_{new}. Except for the seed, a random parameter is taken and modified based on the following rules:

- $c_{new}.o = c_{old}.o \pm 1$
- $c_{new}.t = c_{old}.o \pm x, x \in [-0.1, 0.1]$
- $c_{new}.s_i = c_{old}.s_i \pm x, x \in [-0.1, 0.1]$
- $c_{new}.o_i = c_{old}.o_i \pm x, x \in [-0.5, 0.5]$
- $c_{new}.z = c_{old}.z \pm x, x \in [-0.1, 0.1]$

Except for scale factors and translation offsets, all other parameters are constrained to their original domain of definition, as described in initial population configuration.

Definition of crossover operator

The crossover operator takes two chromosomes, c_1 and c_2, then returns a new one, c_{new}, with parameters chosen randomly given the following constraints:

- $c_{new}.s \in \{c_1.s, c_2.s\}$

- $c_{new}.o = \overline{c_1.o, c_2.o}$
- $c_{new}.t \in [c_1.t, c_2.t]$
- $c_{new}.s_i \in [c_1.s_i, c_2.s_i]$
- $c_{new}.o_i \in [c_1.o_i, c_2.o_i]$
- $c_{new}.z \in [c_1.z, c_2.z]$

Experimental analysis

Research hypothesis

The hypothesis for this study is that a supervised classifier such as the one described in the previous chapter is capable of converging towards an optimal solution. While such a method could prove to be appliable even in the case of regression problems, this paper only tackles the classification problem.

Research questions

- Can such a supervised classifier converge towards an optimal solution?
- If so, what accuracy can it achieve and is it competitive compared to other methods?
- How long does it take to train a model compared to these other methods?
- What about the time necessary to classify new input once training takes place?
- Does this approach generalize to various types of problems, or is it restrained to a particular subset?
- How well does it scale with the number of features in terms of computation time and accuracy?

Experimental procedure

The experimental procedure consists of identifying a supervised classification problem and a suitable, labeled dataset for said problem. Then, depending on size of the dataset, it is to be divided randomly between training data, used to teach the model about the particularities of the problem, and validation data, to evaluate the accuracy of the learnt model.
Both training data and validation data is then normalized using Z normalization fit on the training data only with the purpose of supporting the solution search. Training data is then fed to the classifier to make predictions and adjust the internal parameters as to minimize an error function.
Once trained, the found classifier is tested by comparing the predicted classes for the inputs in the validation dataset with the ones that come from the real data. Other metrics to supplement accuracy are represented by the confusion matrix, recall and precision in context of multiclass labelling.

Original approach

The original approach consists in using the parameters of an *N*-dimensional Perlin noise to encode a predicted solution landscape, which can then further be used to retrieve a binary classification label for an individual input consisting of n features. This supervised, binary classifier approach could allow for trivial reproduction of results, because Perlin noise can generate consistent results given the same pseudo-random number generator seed and parameters. Moreover, class prediction for a new input could be considerably simple to compute, similar to SVMs, while still maintaining a non-linear classification potential, such as that of ANNs.

Description of chosen experiments

For a preliminary experiment, the classifier will be used to model logic gates, such as AND, OR, XOR, similar to the original perceptron. Given the domain-limited amount of data for this problem, acquiring it will be done through synthesis of an exhaustive list of cases, while splitting into training and testing sets will be forgone. The expected outcome is that of 100% accuracy on all 3 situations, the later one being essential for proving the non-linear character of the classifier.
For the second experiment, which will work on Fisher's iris flower dataset [6], it is intended to train a model of the proposed classifier which can distinguish between 3 species of iris flowers: setosa,

virginica and versicolor, given their petal and sepal sizes. Given that the proposed classifier is a binary classifier in nature, the technique used to solve this multiclass classification problem is that of one-versus-all [7], in which a number of classifiers equal to the number of classes in the problem domain will be trained to distinguish between elements that are or are not in each class. The performance of individual classifiers will be compared to that of previous, related works [8], and the performance of the general classifier, as well, with [9], [10] and [11]. In terms of accuracy, a result of 100% would be ideal, one approximatively around 97% would be competitive, while above 90% it would be satisfactory, and above 80%, sufficient. A result below this threshold would argue against the use of a classifier such as the one proposed, without any modifications.

Study case

Experiment 1: Logic gates

In this experiment, the classifier will be trained to model the following logic gates: AND, OR and XOR. This will serve as a preliminary test to the viability of the proposed model as a classifier that can adapt to given data. For this purpose, the desired accuracy will be of 100%, given that existing methods can already learn such simple models without any problems.
The data will be obtained by manually completing CSV input files with the truth value tables, one CSV file per logic gate. The resulting files will have the following structure:

AND			**OR**			**XOR**		
x	**y**	**result**	**x**	**y**	**result**	**x**	**y**	**result**
0	0	0	0	0	0	0	0	0
0	1	0	0	1	1	0	1	1
1	0	0	1	0	1	1	0	1
1	1	1	1	1	1	1	1	0

Table 1 Truth values for AND, OR and XOR logic gates

This data is loaded into the program, without being split into training and validation datasets, given the small size and requirement for perfect accuracy. The fitness function of the genetic algorithm will then use this data to measure the accuracy of the trained classifier.
Upon running the genetic algorithm with a population size and number of iterations equal to 10, a solution with an accuracy measure of 1 should result, which will then be displayed visually for manual inspection.

AND logic gate

The results of running the genetic algorithm on data for AND logic gate are as follow:

```
Best solution is x = [98396533, 1, 0.6831648088645714, 1.84863487558408,
5.09674368033715, -2.068009755082906, 2.7965782520191254,
0.9235185212512687]
f(x) = 0.0
```

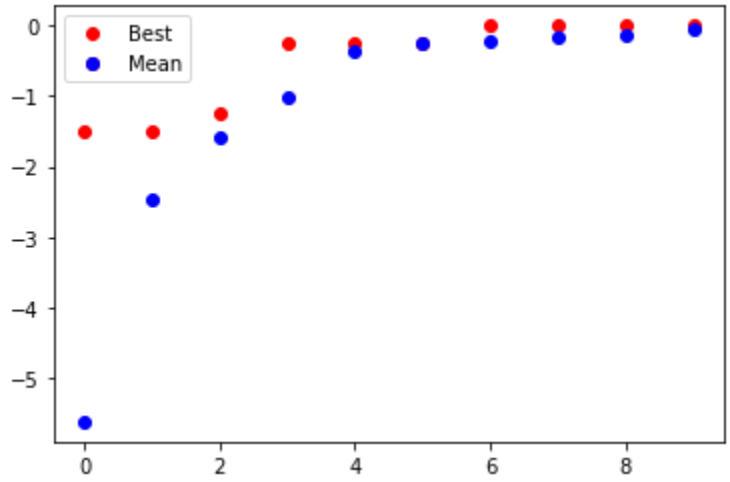

Figure 6 Evolution graph of mean and best fitness in populations across generations in AND logic gate learning (x = generation, y = fitness)

In this case, $f(x) = acc(x) - x.o$, where $acc: C \rightarrow [0,1]$ is a function that computes the accuracy of classifier x. From the chromosome representation, it is observed that $x.o = 1$, and given that $f(x) = 0$, it is implied that the accuracy was equal to 1, or 100%, which validates the AND logic gate case. As expected of the steady state evolution tactic, the results increase in a monotonous manner, the average fitness tending towards the best fitness as the generation count increases.
The visualization of the obtained classifier and the original Perlin noise are as follow:

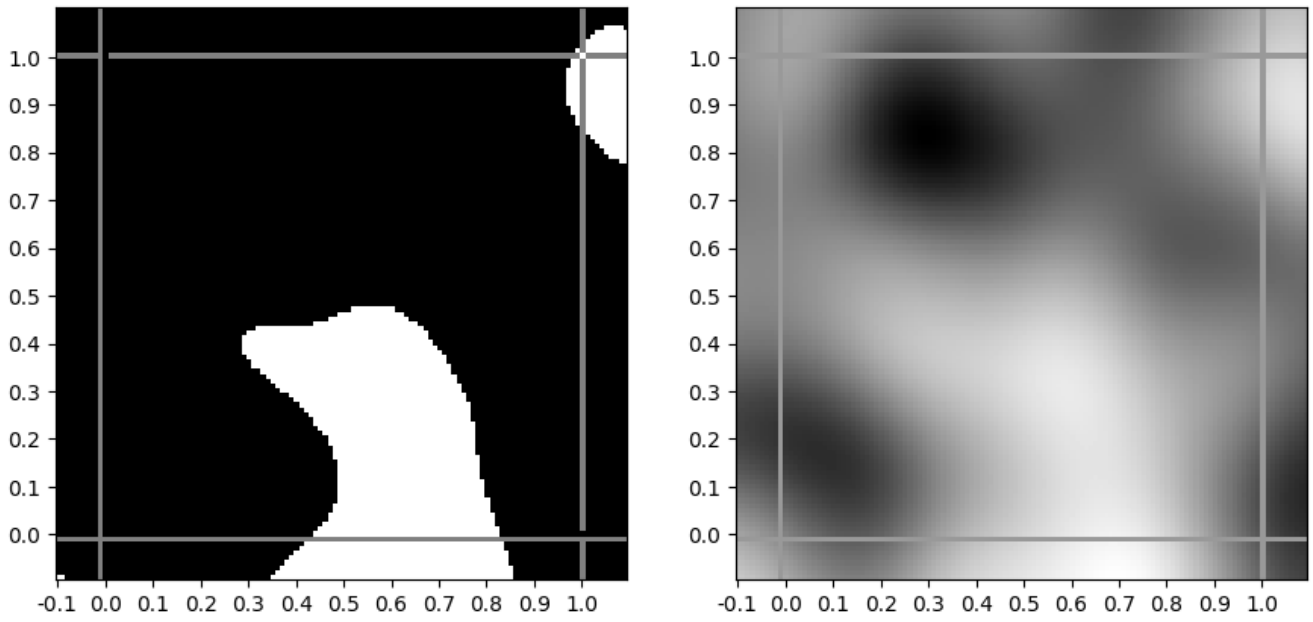

Figure 7 Obtained AND classifier and original Perlin noise graph

It can be observed that in the lack of samples to constrain the results within the input interval $(0,1)^2$, the classifier adopts a random class based on the properties of the underlying Perlin noise, which might not prove to be an optimal solution in many situations. Moreover, this can later lead to overlapping regions in one-versus-all classifiers, which are to be solved by imposing a strict order of evaluation.

OR logic gate

The results for this case are as follow:

```
Best solution is x = [76617226, 1, 0.4117759140208105, -7.424920819460869,
4.169411694848429, -1.0178520425969708, -0.4684833669767699, -
1.9511380584580513]
f(x) = 0.0
```

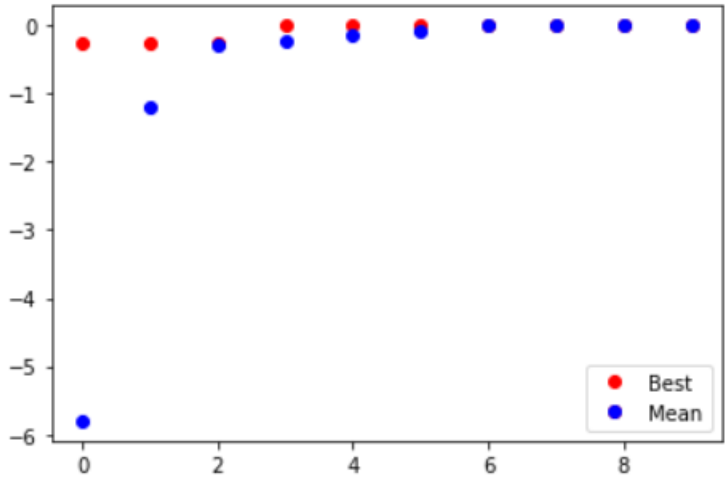

Figure 8 Evolution graph of mean and best fitness in populations across generations in OR logic gate learning (x = generation, y = fitness)

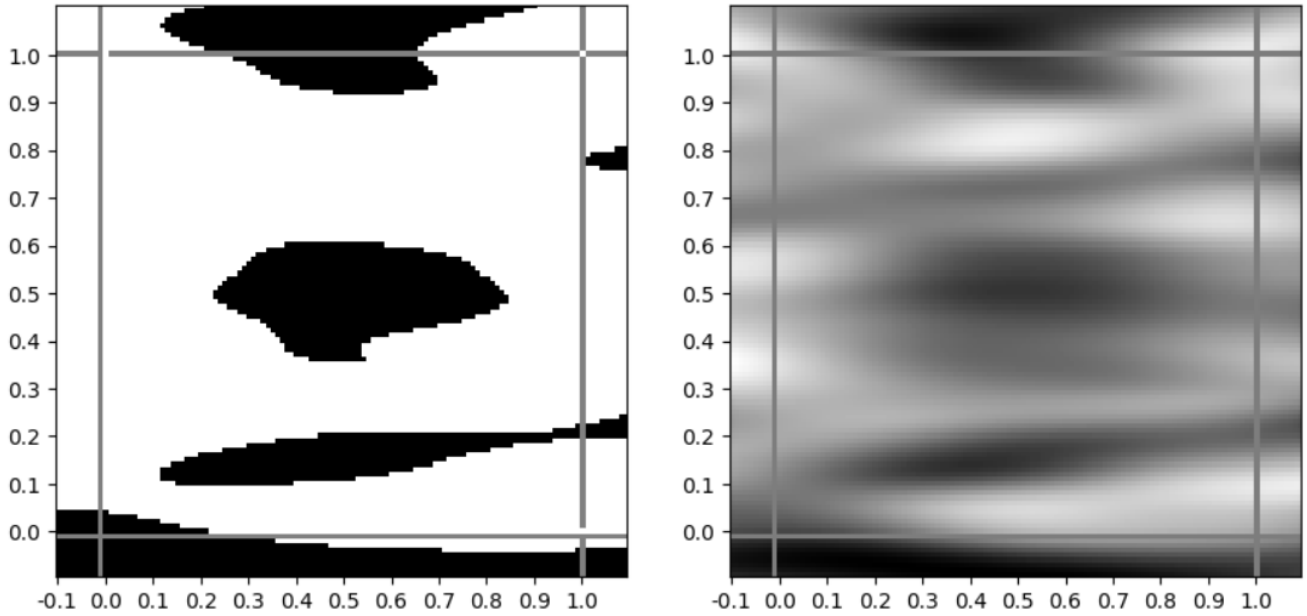

Figure 9 Obtained OR classifier and original Perlin noise graph

It can be observed that even in the case of simple solution landscapes, the proposed classifier is not necessarily similarly simple. This is due to the same reason as aforementioned in the case of the AND classifier, and namely, due to the lack of constraining input samples. This can be solved by inserting additional criteria into the fitness function, such as the proposed turbulence minimization or minimizing scale as well.

XOR logic gate

The results for this case are as follow:

```
Best solution is x = [41347066, 1, 0.7458423618591936,
0.032928970047108486, -4.146307430501202, -0.08782782684939995,
4.7302458478462945, 3.913304420239831]
f(x) = 0.0
```

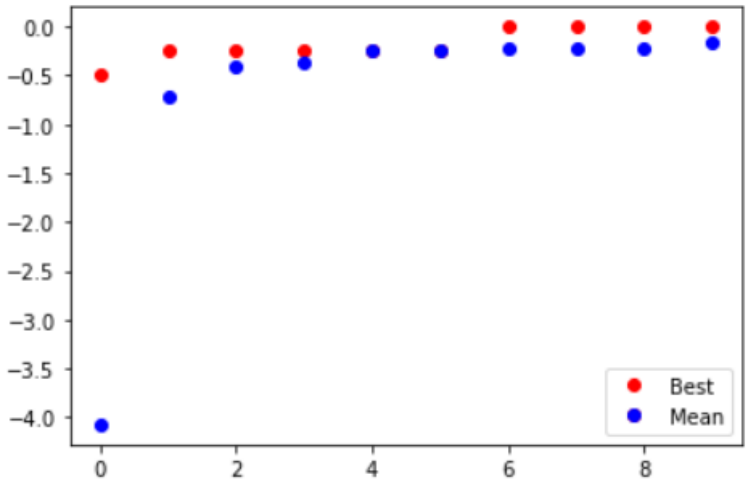

Figure 10 Evolution graph of mean and best fitness in populations across generations in XOR logic gate learning (x = generation, y = fitness)

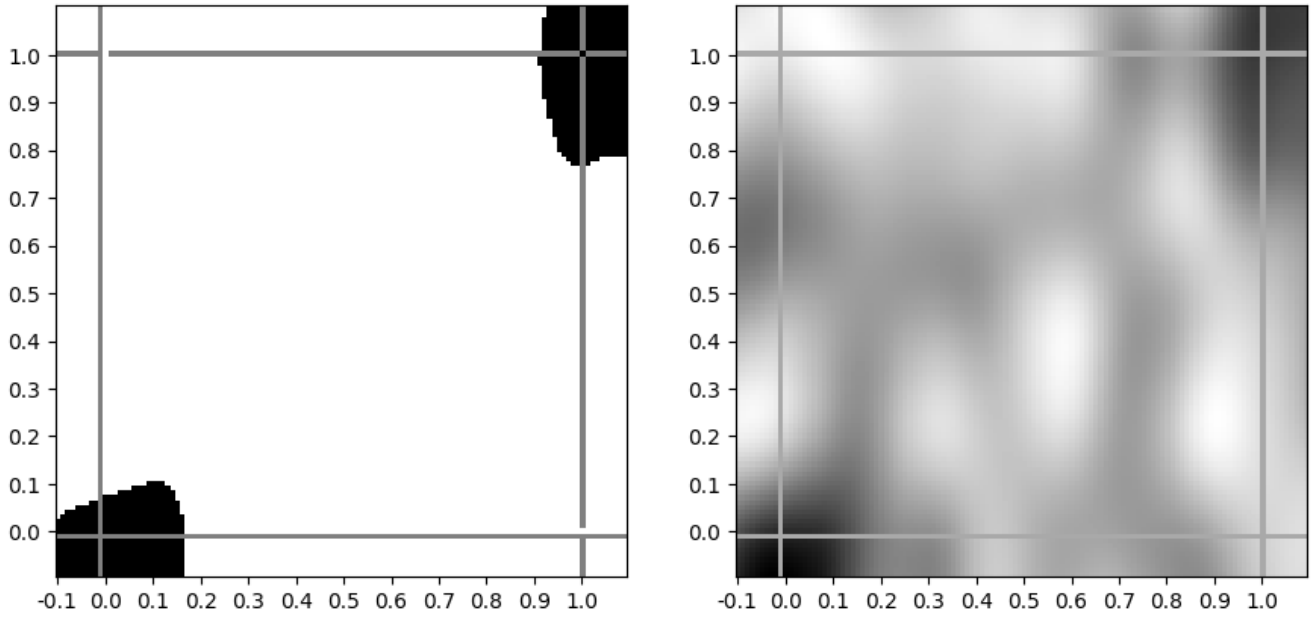

Figure 11 Obtained XOR classifier and original Perlin noise graph

The results above confirm the accuracy expected of the proposed classifier and prove the non-linear character that it possesses.

Experiment 2: Iris flower dataset

In this experiment, three classifiers will be trained to distinguish between each of the 3 species of iris

flowers: iris setosa, iris versicolor and iris virginica. The data is obtained from Fisher's iris flower dataset [6] as a CSV file, which is then loaded into the program, where the fitness function will make use of it for validation of classifier accuracy. In this scenario, the fitness function does not contain constraints related to the shape of the underlying Perlin noise.
This dataset is often used in artificial intelligence and machine learning domains as a reference problem for testing. As such, the proposed classifier will be tested under the same conditions presented in other works ([8], [9], [10], [11]) for the purpose of comparison. Namely, the difference in the enumerated works is represented by the ratio by which the data is split into training and validation data, such as either 50%, 66%, 80% or a presumed 100% (in the case of [9]) samples being assigned to training data. It was observed by empirical means that training the classifiers on all the available input features not only results in a considerably longer training duration, but also in poorer results [12]. As such, the features chosen for input classification are petal lengths and petal widths, for which the data is distributed as pictured.

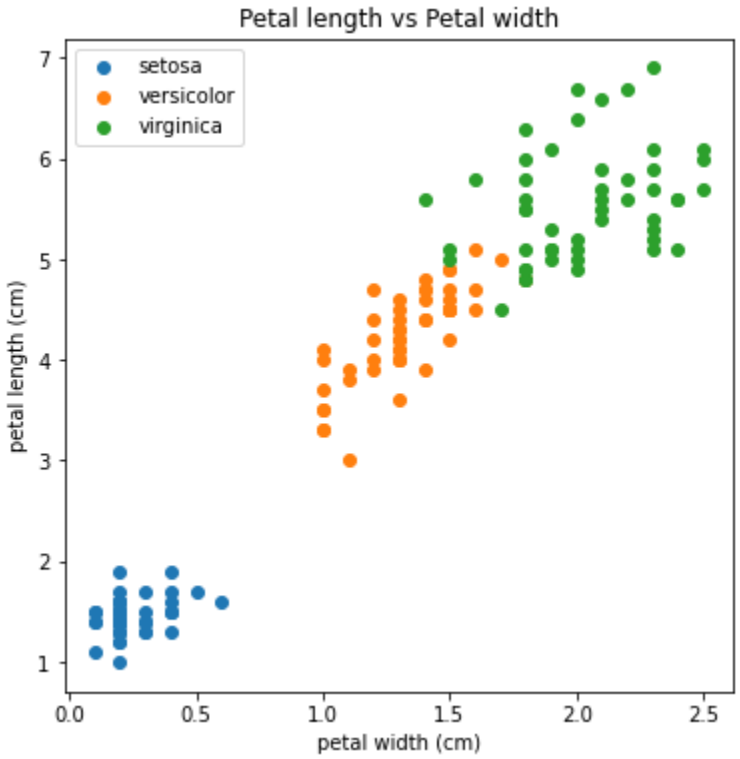

Figure 12 Distribution of data in the iris flower dataset

The genetic algorithm will be working with a population size and generation count of 100.
Once all three classifiers are trained, each of them is checked (in the order: Iris-setosa, Iris-versicolor, Iris-virginica) until a class is assigned to the input. If no label is assigned at the end, it could either be considered as part of the last class, case in which the last classifier is not necessary. However, to better assess the accuracy of the learnt models, unlabeled inputs are left as such.
The results for each case and classifier are shown in the following part.

Case 1: 50% training data

The data was split in 50% training data and 50% validation data.

Iris setosa classifier

```
Best solution is x = [83387065, 4, 0.7435772906167105,
-0.11734515906277565, 2.0518830714384095, -0.011135938566580683,
-0.8926898760419555, -4.114093714111469]
f(x) = 1.0
```

As shown here, the classifier has managed to attain an accuracy of 100% for the linearly separable class.

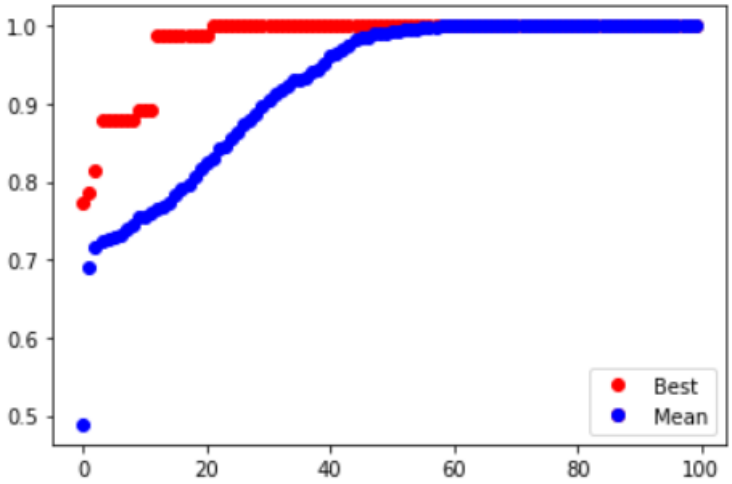

Figure 13 Evolution graph in Iris setosa classifier learning (50% training data distribution)

The solution has managed to converge to an optimal configuration in a reasonable number of generations, and with this occasion we can once again see more clearly the monotonous increase of the fitness function value.

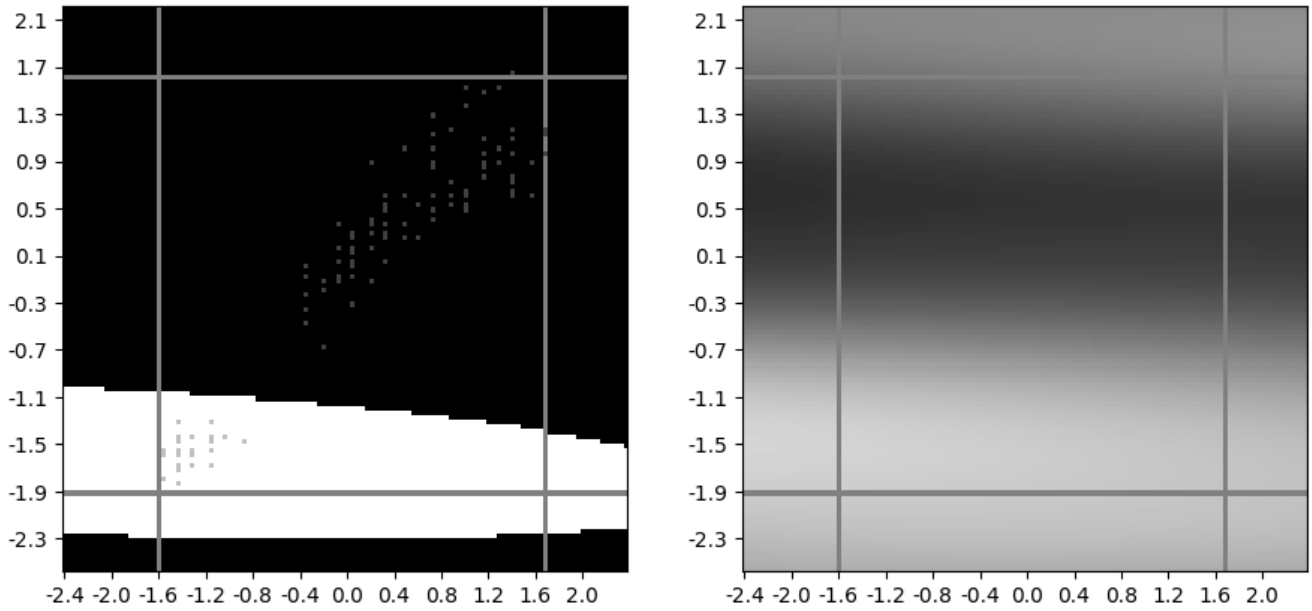

Figure 14 Obtained Iris setosa classifier and original Perlin noise graph (50% training data distribution)

Iris versicolor classifier

```
Best solution is x = [74071088, 3, 0.812779545327969, -0.20156798382437985,
5.832052732597587, -0.08959266943392134, -0.06853832485686523,
-3.5600630665944433]
f(x) = 0.9733333333333334
```

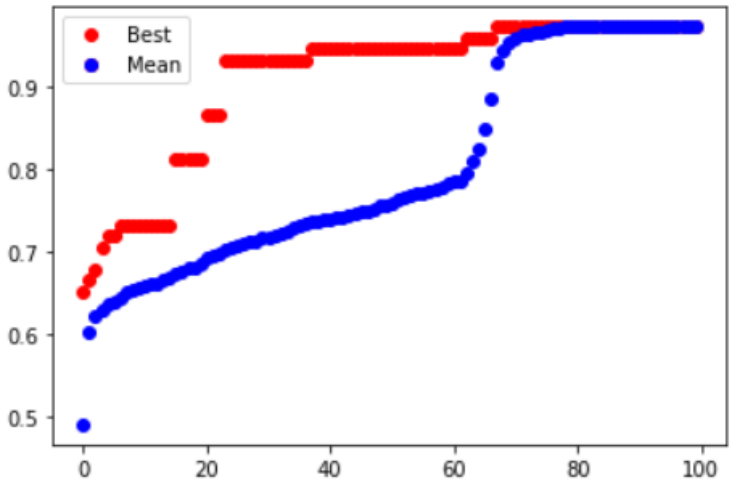

Figure 15 Evolution graph in Iris versicolor classifier learning (50% training data distribution)

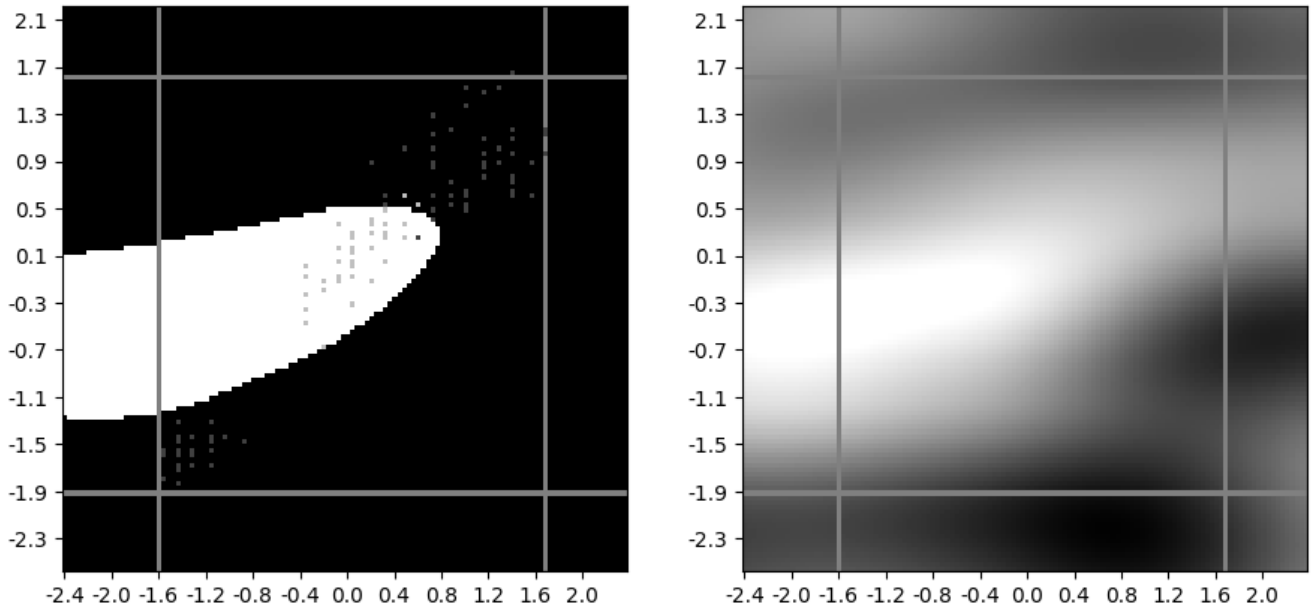

Figure 16 Obtained Iris versicolor classifier and original Perlin noise graph (50% training data distribution)

Iris virginica classifier

```
Best solution is x = [54291783, 1, 0.7041157432109316,
-0.18715212059185923, 0.016621861178946484, 0.27592054481486267,
1.7053073833958625, 1.471582253961968]
f(x) = 0.9733333333333334
```

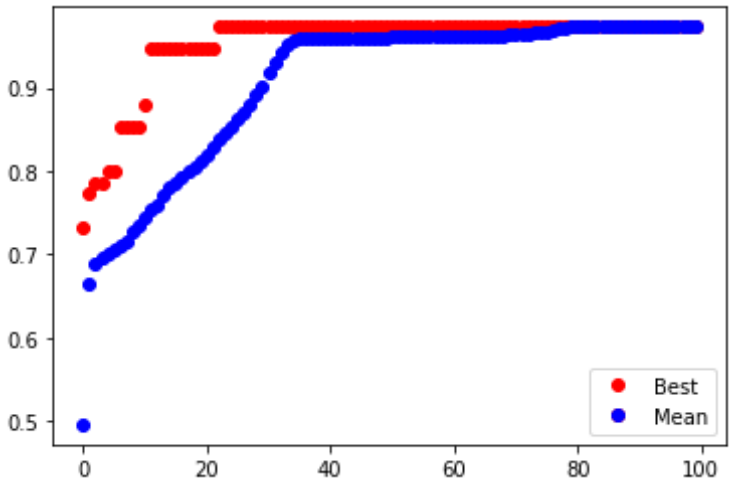

Figure 17 Evolution graph in Iris virginica classifier learning (50% training data distribution)

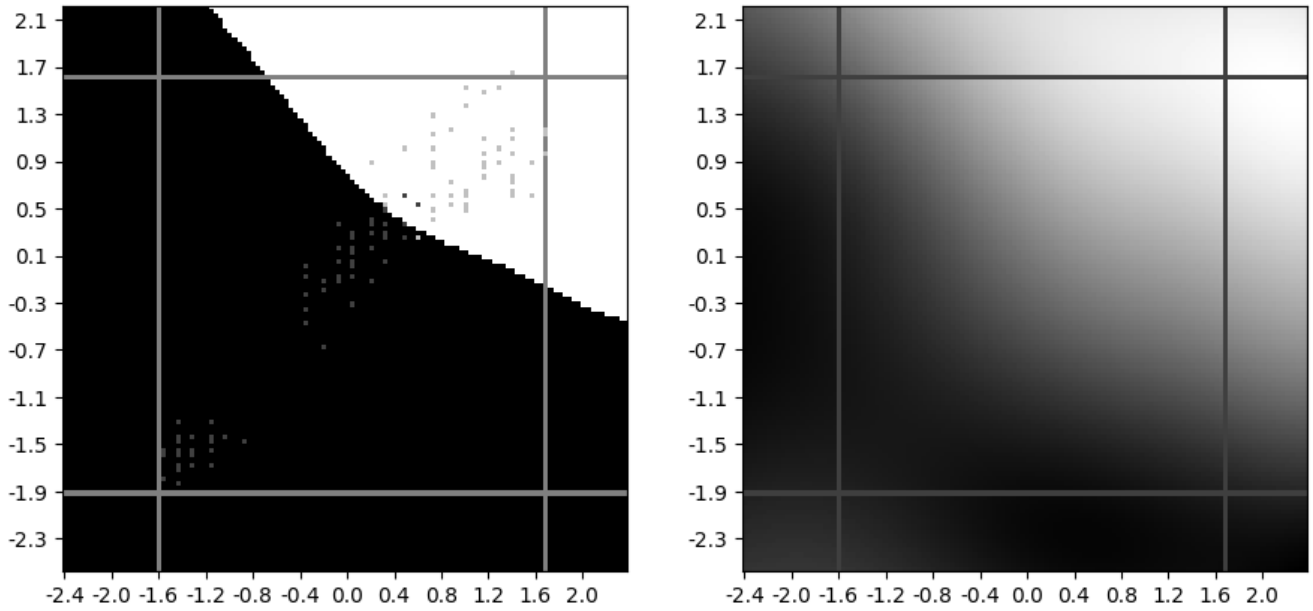

Figure 18 Obtained Iris virginica classifier and original Perlin noise graph (50% training data distribution)

Overall results

```
Classifier validation accuracy: 0.96
Classifier validation precisions: {'Iris-setosa': 1.0, 'Iris-versicolor':
0.9090909090909091, 'Iris-virginica': 0.96}
Classifier validation recalls: {'Iris-setosa': 1.0, 'Iris-versicolor':
0.9523809523809523, 'Iris-virginica': 0.9230769230769231}

Classifier training accuracy: 0.9733333333333334
Classifier training precisions: {'Iris-setosa': 1.0, 'Iris-versicolor':
1.0, 'Iris-virginica': 0.9230769230769231}
Classifier training recalls: {'Iris-setosa': 1.0, 'Iris-versicolor':
0.9310344827586207, 'Iris-virginica': 1.0}

Classifier overall accuracy: 0.9666666666666667
Classifier overall precisions: {'Iris-setosa': 1.0, 'Iris-versicolor':
0.9591836734693877, 'Iris-virginica': 0.9411764705882353}
Classifier overall recalls: {'Iris-setosa': 1.0, 'Iris-versicolor': 0.94,
'Iris-virginica': 0.96}
```

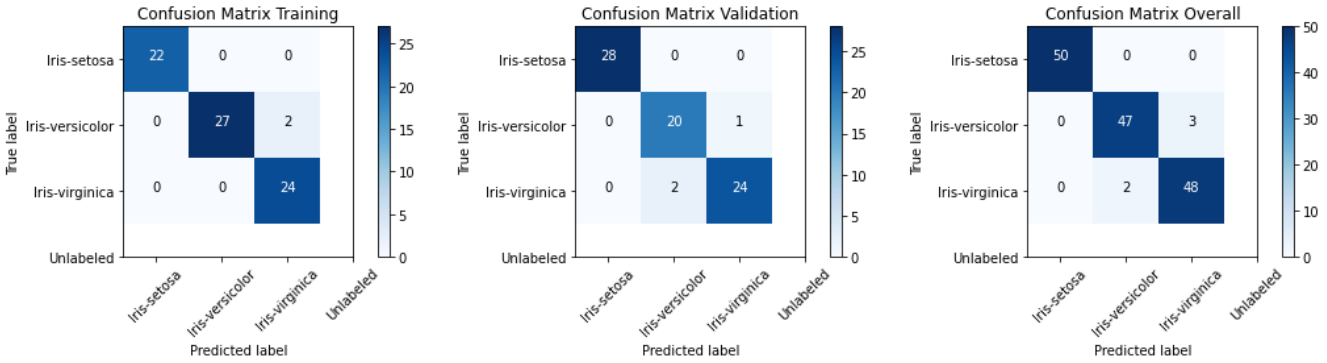

Figure 19 Training, validation and overall confusion matrices for iris flower classification (50% training data distribution)

Case 2: 66% training data

The data was split in two thirds training data and one third validation data.

Iris setosa classifier

```
Best solution is x = [32236971, 3, 0.7369063231728004, -0.3821472507821688,
-4.135814046664476, 0.070340084008226, 0.958272895629876,
0.2118536544416667]
f(x) = 1.0
```

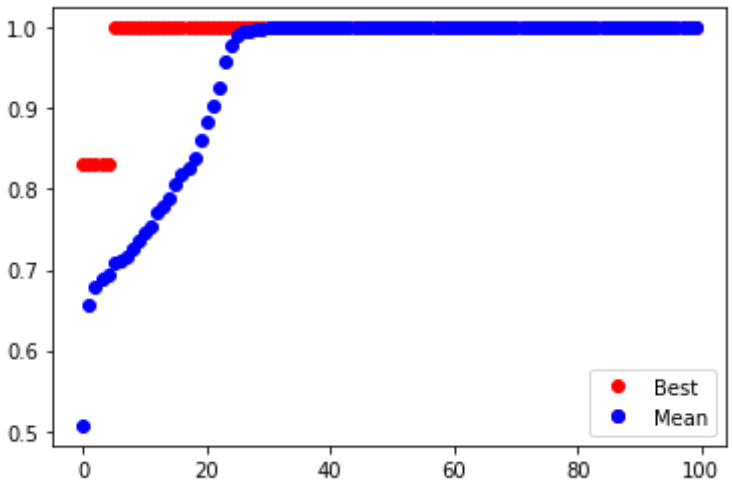

Figure 20 Evolution graph in Iris setosa classifier learning (66% training data distribution)

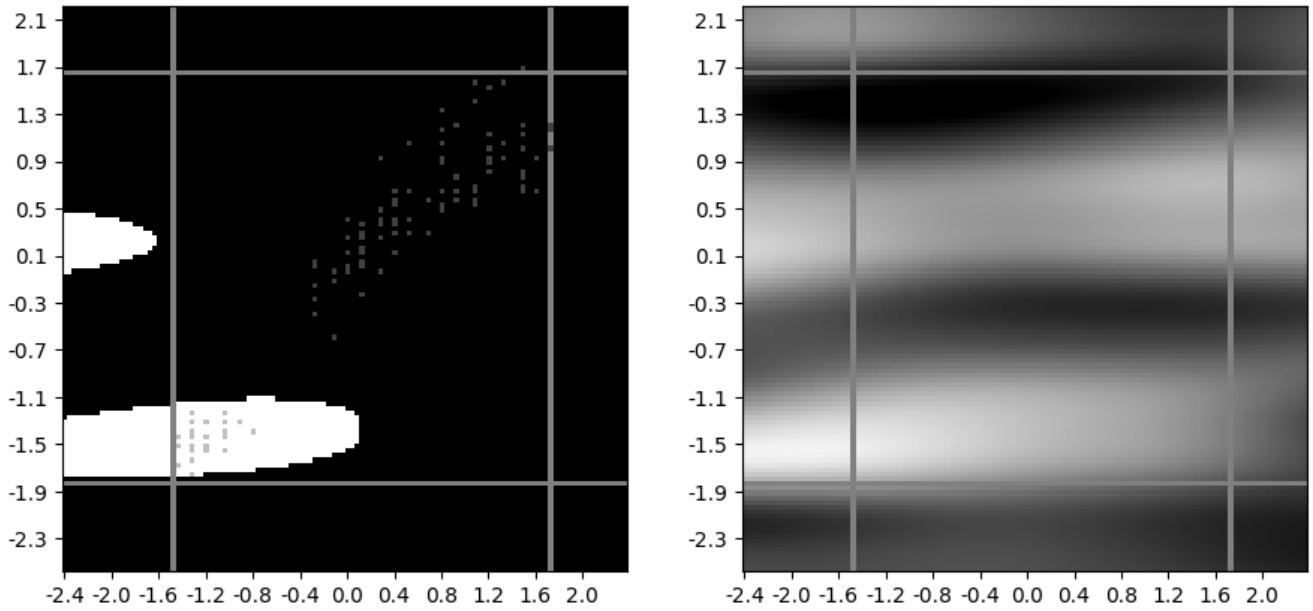

Figure 21 Obtained Iris setosa classifier and original Perlin noise graph (66% training data distribution)

Iris versicolor classifier

```
Best solution is x = [3291003, 1, 0.703697722643939, -0.21723082318425116,
-1.4901729986179468, -1.3700566968391208, 1.1629306696830266,
2.1401928670917147]
f(x) = 0.99
```

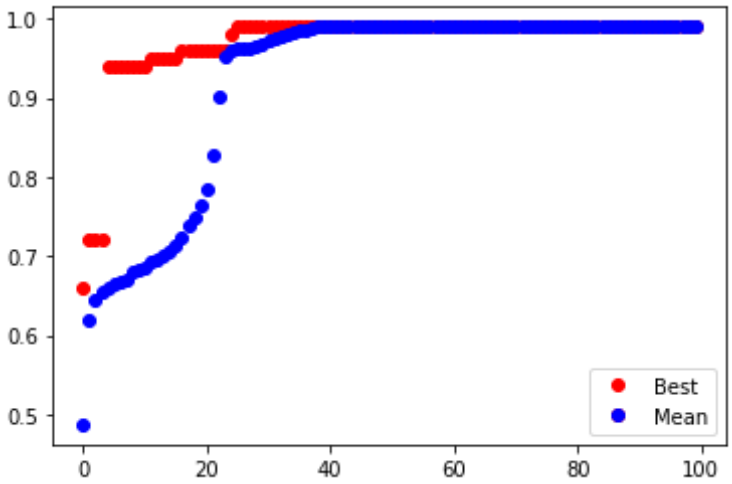

Figure 22 Evolution graph in Iris versicolor classifier learning (66% training data distribution)

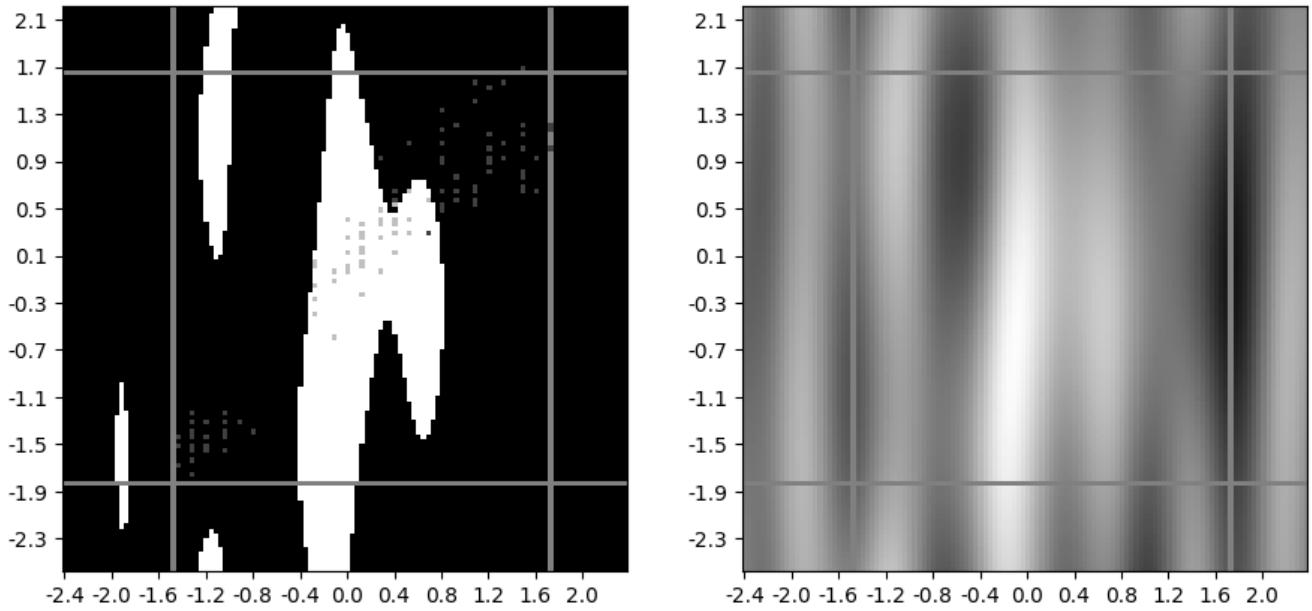

Figure 23 Obtained Iris versicolor classifier and original Perlin noise graph (66% training data distribution)

Iris virginica classifier

```
Best solution is x = [75941276, 3, 0.6681402375086566, -
0.08363995932751829, 2.2337465708319897, 0.23921549939045503, -
2.2427225966158812, -0.896376835841215]
f(x) = 0.98
```

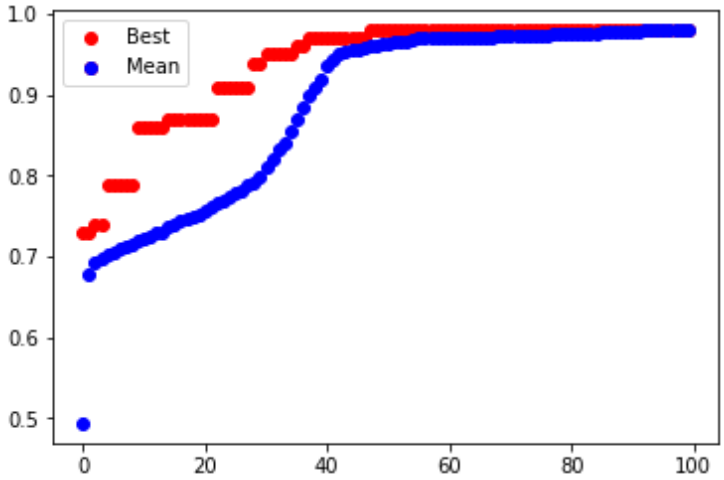

Figure 24 Evolution graph in Iris virginica classifier learning (66% training data distribution)

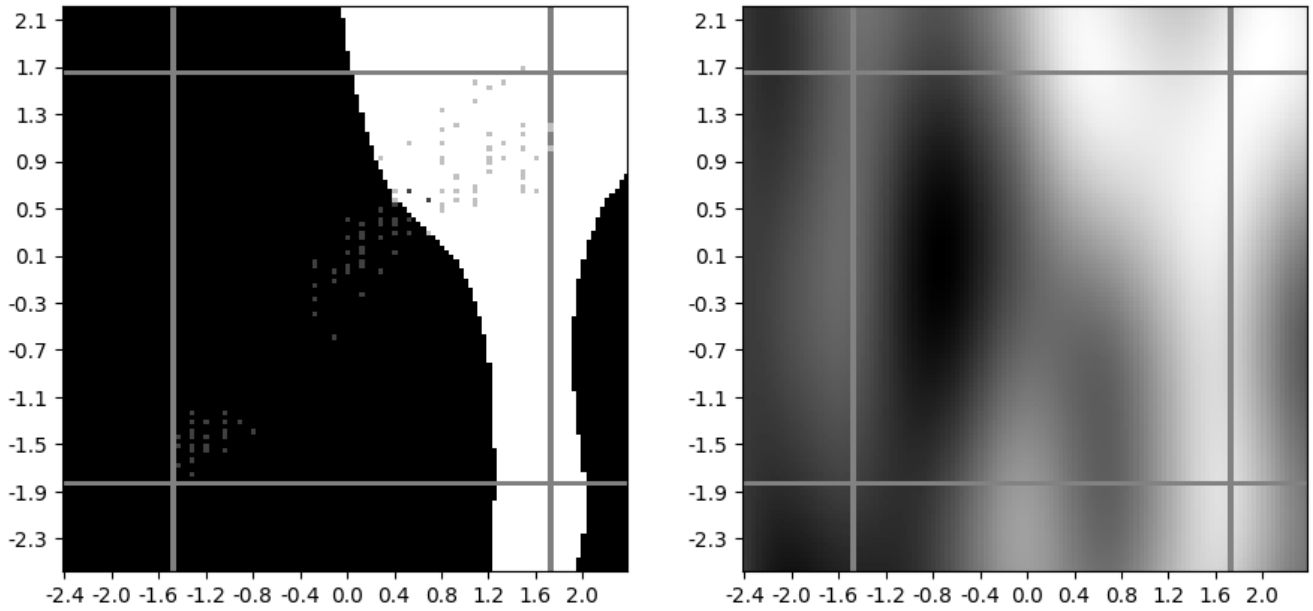

Figure 25 Obtained Iris virginica classifier and original Perlin noise graph (66% training data distribution)

Overall results

```
Classifier validation accuracy: 0.98
Classifier validation precisions: {'Iris-setosa': 1.0, 'Iris-versicolor':
0.9333333333333333, 'Iris-virginica': 1.0}
Classifier validation recalls: {'Iris-setosa': 1.0, 'Iris-versicolor': 1.0,
'Iris-virginica': 0.9473684210526315}

Classifier training accuracy: 0.99
Classifier training precisions: {'Iris-setosa': 1.0, 'Iris-versicolor':
1.0, 'Iris-virginica': 0.96875}
Classifier training recalls: {'Iris-setosa': 1.0, 'Iris-versicolor':
0.9722222222222222, 'Iris-virginica': 1.0}

Classifier overall accuracy: 0.9866666666666667
Classifier overall precisions: {'Iris-setosa': 1.0, 'Iris-versicolor':
0.98, 'Iris-virginica': 0.98}
Classifier overall recalls: {'Iris-setosa': 1.0, 'Iris-versicolor': 0.98,
'Iris-virginica': 0.98}
```

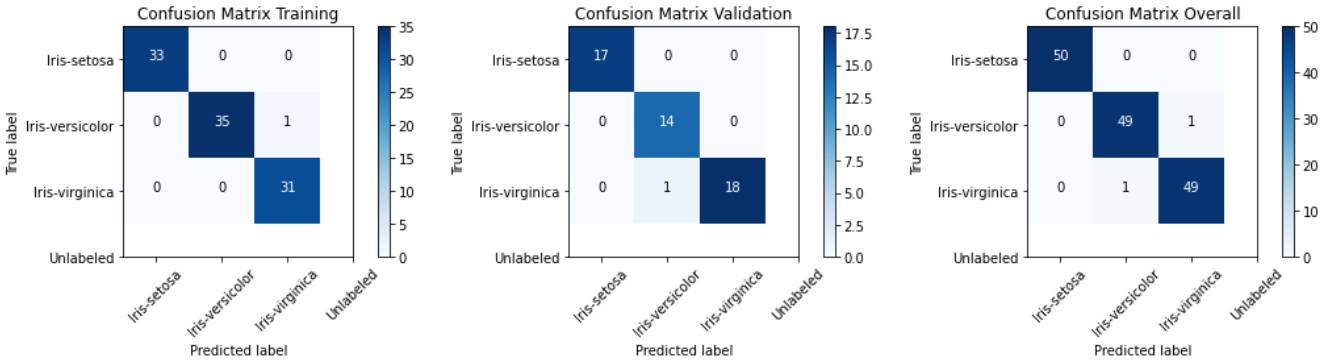

Figure 26 Training, validation and overall confusion matrices for iris flower classification (66% training data distribution)

Case 3: 80% training data

The data was split in 80% training data and 20% validation data.

Iris setosa classifier

```
Best solution is x = [41899788, 4, 0.7263264983817371, -
0.08081986503364569, -1.822298514803577, 0.07453857566353328,
1.8674506742139305, 1.1040050801938324]
f(x) = 1.0
```

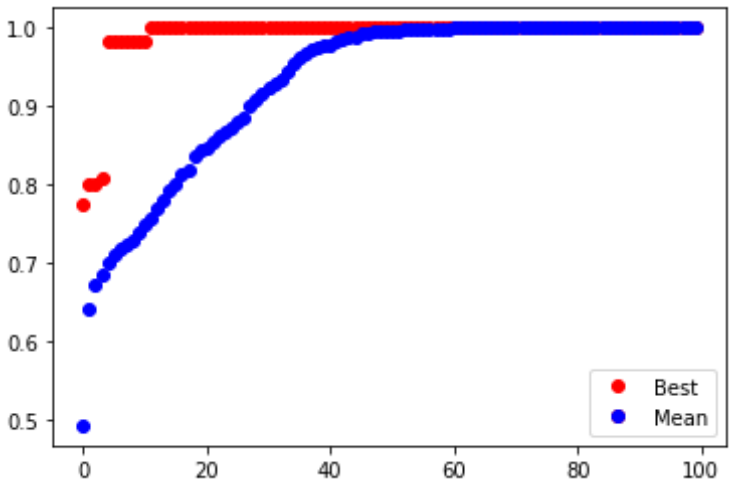

Figure 27 Evolution graph in Iris setosa classifier learning (80% training data distribution)

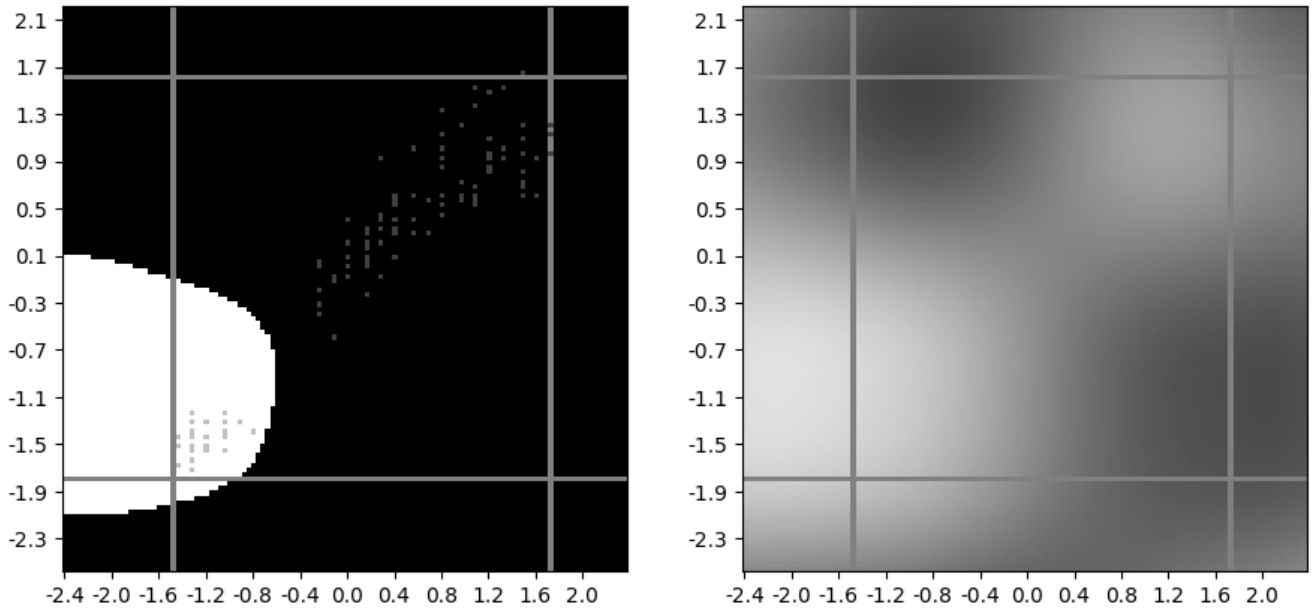

Figure 28 Obtained Iris setosa classifier and original Perlin noise graph (80% training data distribution)

Iris versicolor classifier

```
Best solution is x = [5997008, 1, 0.7091936394481007, 0.8574885773310598,
1.3691810167882614, 0.4970817276143978, 0.6959215956836232,
-2.600011330833613]
f(x) = 0.975
```

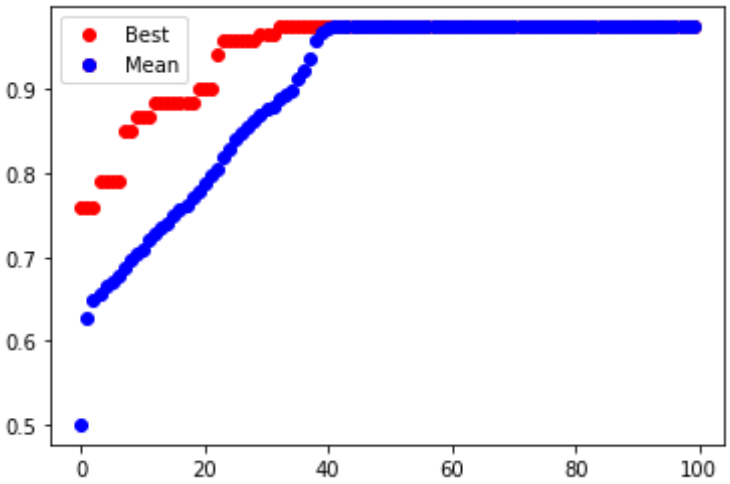

Figure 29 Evolution graph in Iris versicolor classifier learning (80% training data distribution)

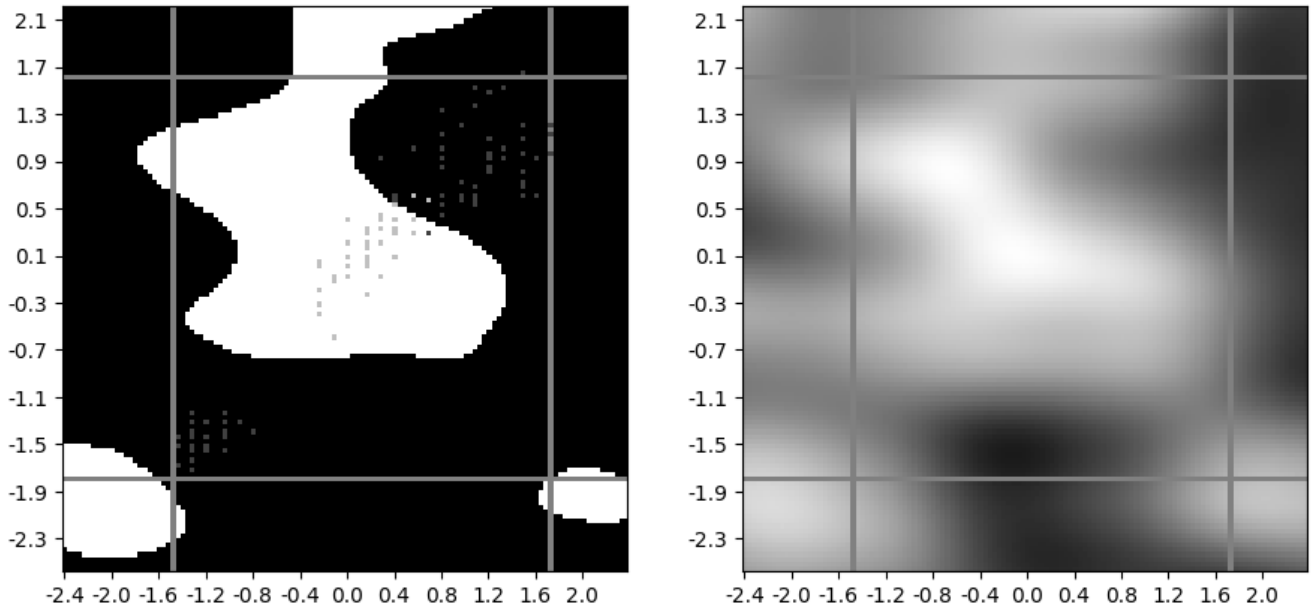

Figure 30 Obtained Iris versicolor classifier and original Perlin noise graph (80% training data distribution)

Iris virginica classifier

```
Best solution is x = [91369968, 2, 0.6373262914615909,
-0.25844614192088167, -1.266112886386261, -0.12021941222913707,
-4.16100170280582, -3.7740665760601564]
f(x) = 0.975
```

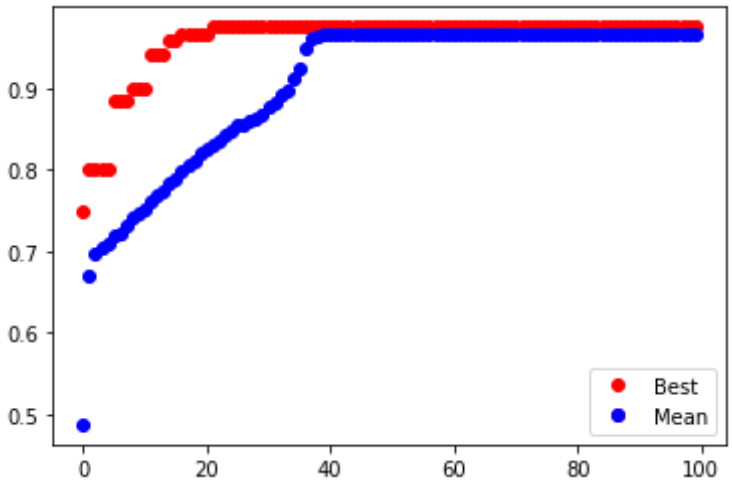

Figure 31 Evolution graph in Iris virginica classifier learning (80% training data distribution)

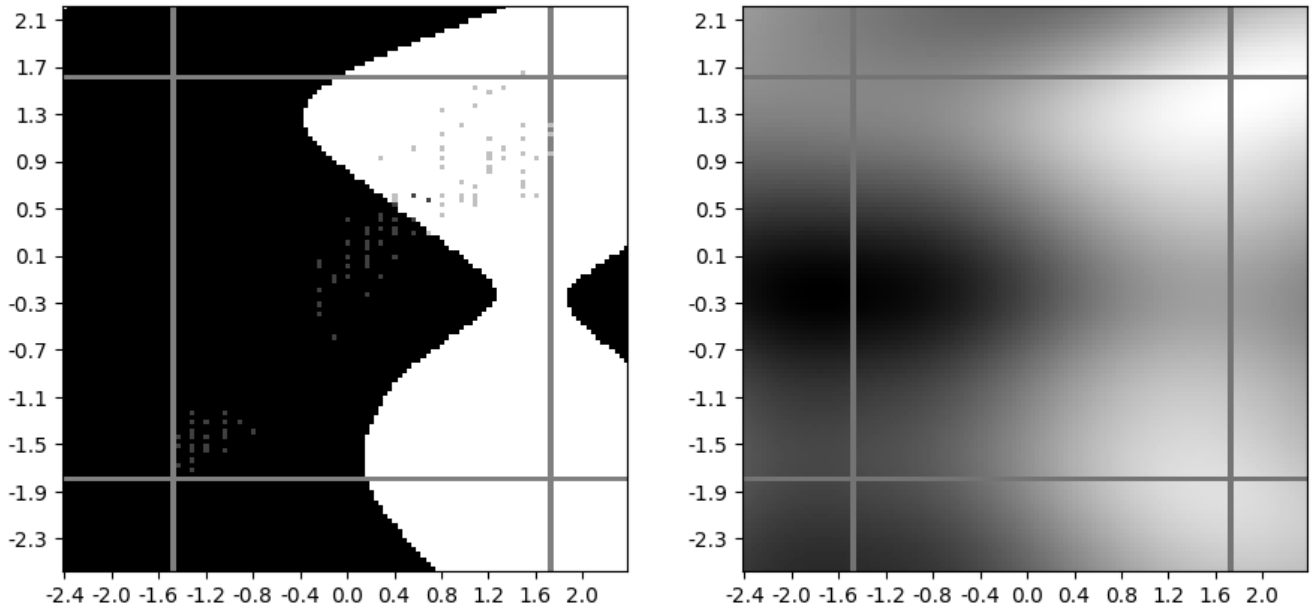

Figure 32 Obtained Iris virginica classifier and original Perlin noise graph (80% training data distribution)

Overall results

```
Classifier validation accuracy: 0.9666666666666667
Classifier validation precisions: {'Iris-setosa': 1.0, 'Iris-versicolor':
0.875, 'Iris-virginica': 1.0}
Classifier validation recalls: {'Iris-setosa': 1.0, 'Iris-versicolor': 1.0,
'Iris-virginica': 0.9230769230769231}

Classifier training accuracy: 0.975
Classifier training precisions: {'Iris-setosa': 1.0, 'Iris-versicolor':
1.0, 'Iris-virginica': 0.925}
Classifier training recalls: {'Iris-setosa': 1.0, 'Iris-versicolor':
0.9302325581395349, 'Iris-virginica': 1.0}

Classifier overall accuracy: 0.9733333333333334
Classifier overall precisions: {'Iris-setosa': 1.0, 'Iris-versicolor':
0.9791666666666666, 'Iris-virginica': 0.9423076923076923}
Classifier overall recalls: {'Iris-setosa': 1.0, 'Iris-versicolor': 0.94,
'Iris-virginica': 0.98}
```

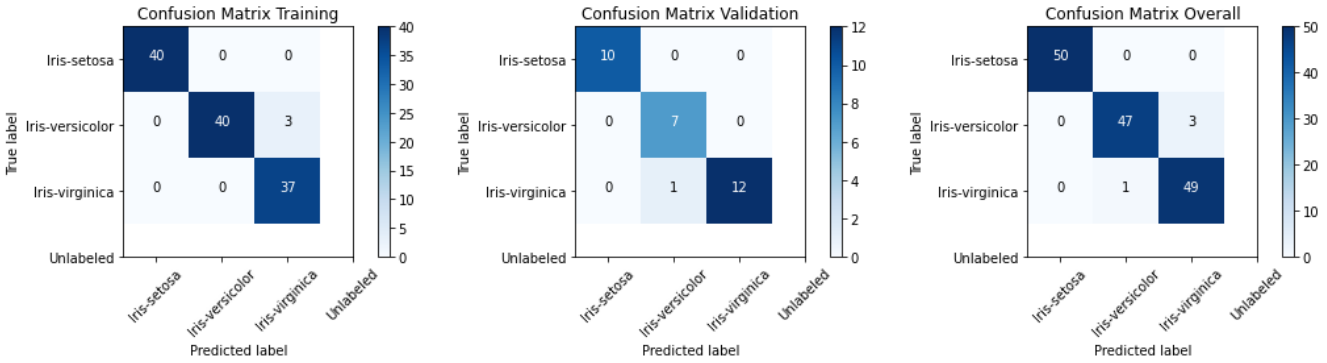

Figure 33 Training, validation and overall confusion matrices for iris flower classification (80% training data distribution)

Case 4: 100% training data

The data was not split into training and validation datasets, but fed directly to the learning algorithm. This is however not recommended, as it runs the risk of obtaining an overfitting model.

Iris setosa classifier

```
Best solution is x = [22834589, 1, 0.7814497767379254, 0.38404819548540947,
-0.18943531325629792, 0.26738083693992065, -0.9983630341717837, -
2.1861073803318742]
f(x) = 1.0
```

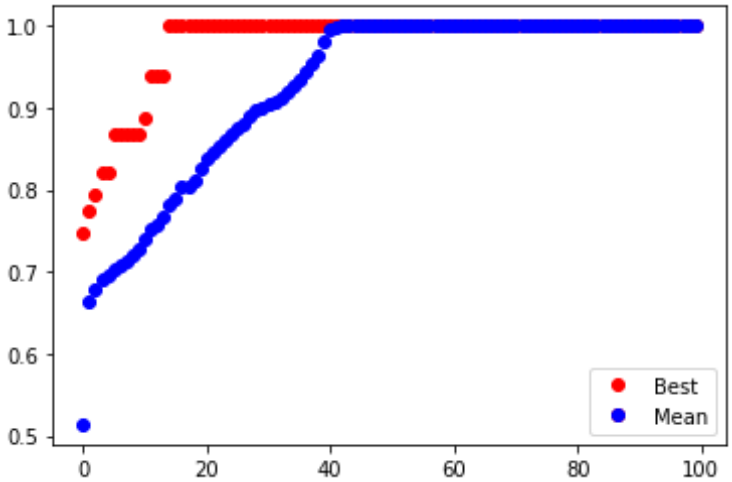

Figure 34 Evolution graph in Iris setosa classifier learning (100% training data distribution)

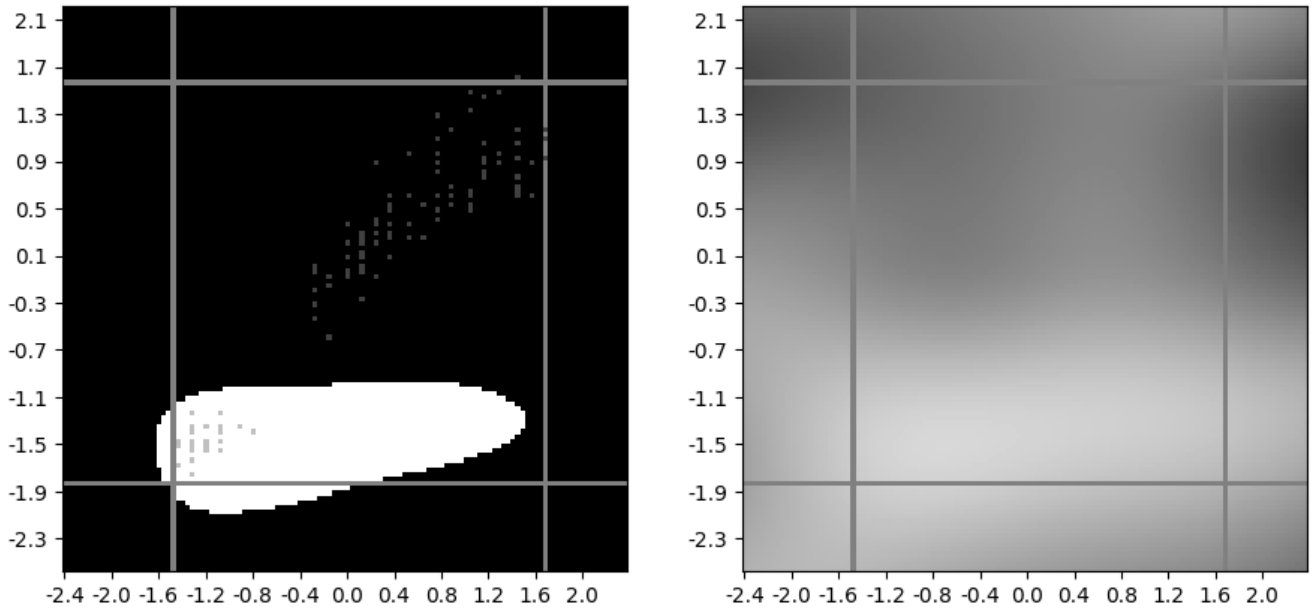

Figure 35 Obtained Iris setosa classifier and original Perlin noise graph (100% training data distribution)

Iris versicolor classifier

```
Best solution is x = [69070805, 6, 0.6814913252651562, -
0.052982379240393146, -2.6931657544040246, 0.06143569759135327, -
3.1009184900866584, 1.4886202371819506]
f(x) = 0.98
```

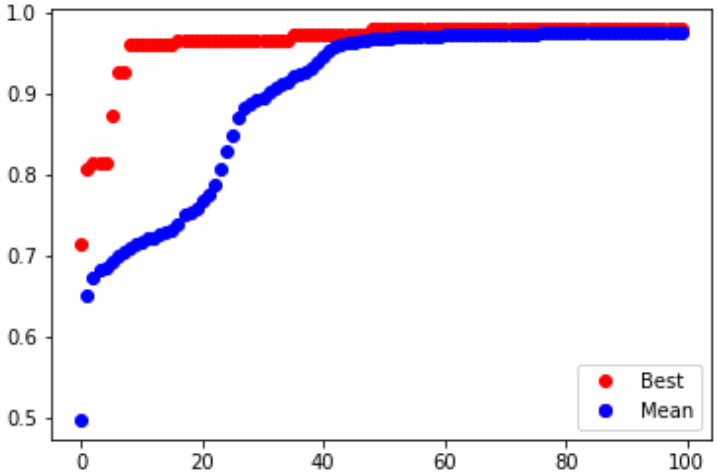

Figure 36 Evolution graph in Iris versicolor classifier learning (100% training data distribution)

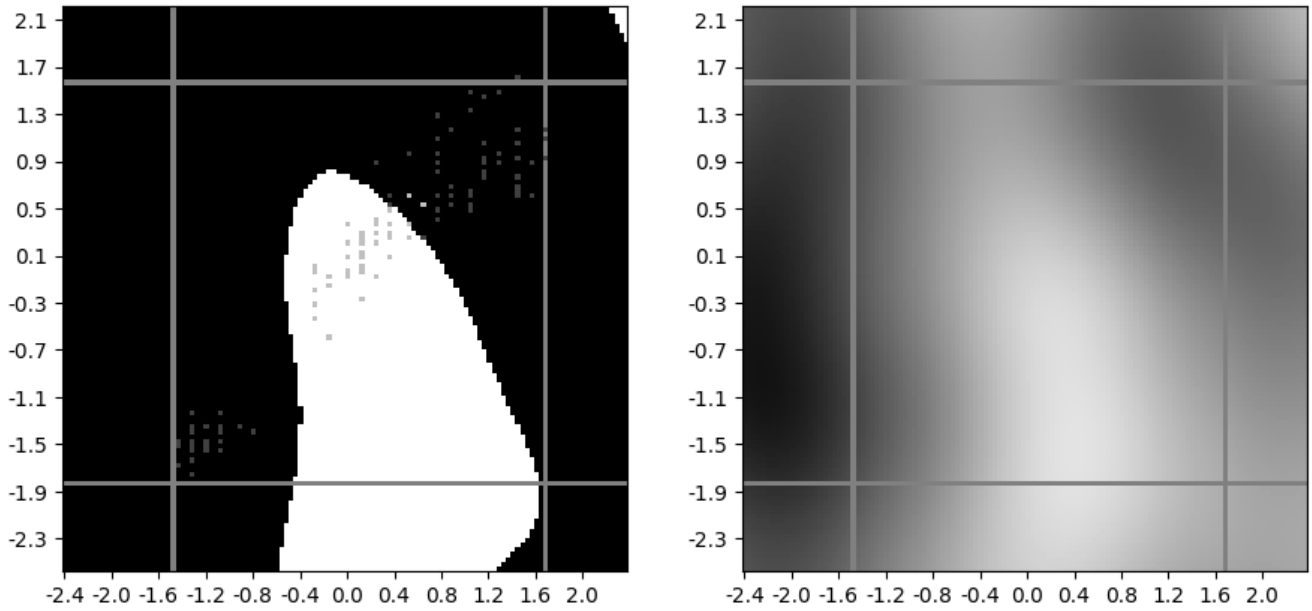

Figure 37 Obtained Iris versicolor classifier and original Perlin noise graph (100% training data distribution)

Iris virginica classifier

```
Best solution is x = [48597780, 1, 0.6472833143037539, -0.510133196805576,
2.4007078064857748, 0.3131887734919495, -0.9477137867699384,
1.9706280676389145]
f(x) = 0.98
```

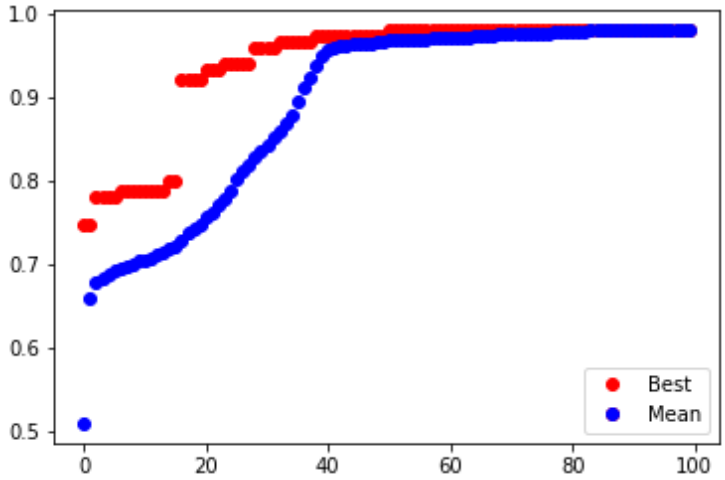

Figure 38 Evolution graph in Iris virginica classifier learning (100% training data distribution)

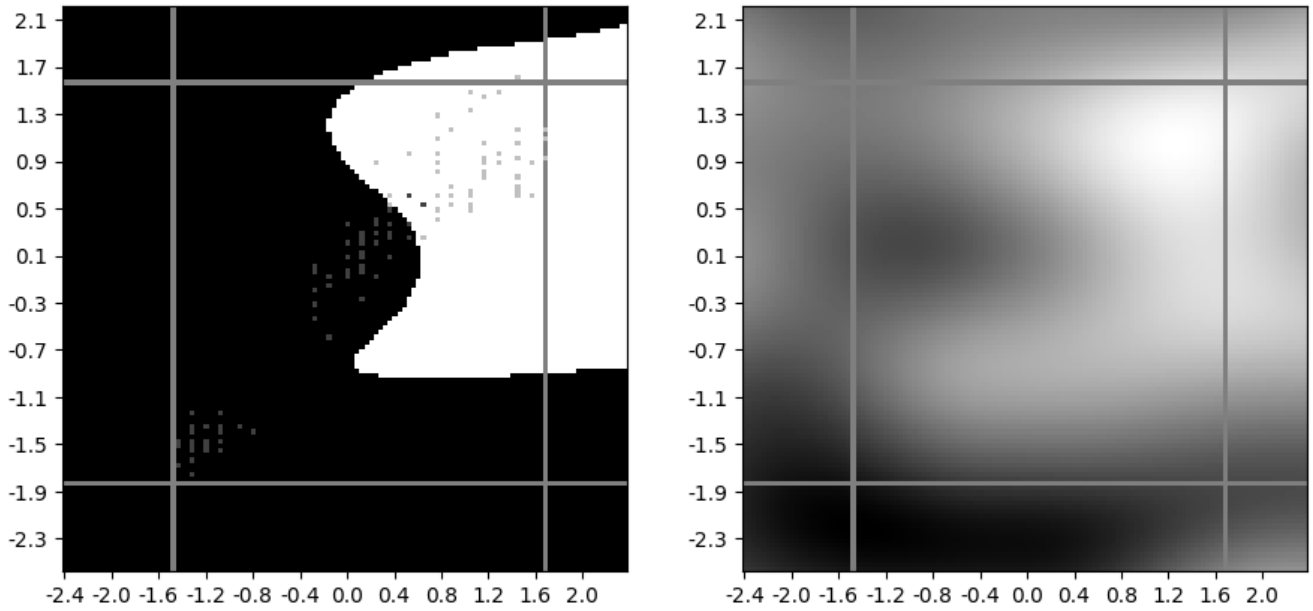

Figure 39 Obtained Iris virginica classifier and original Perlin noise graph (100% training data distribution)

Overall results

```
Classifier overall accuracy: 0.98
Classifier overall precisions: {'Iris-setosa': 1.0, 'Iris-versicolor': 1.0,
'Iris-virginica': 0.9433962264150944}
Classifier overall recalls: {'Iris-setosa': 1.0, 'Iris-versicolor': 0.94,
'Iris-virginica': 1.0}
```

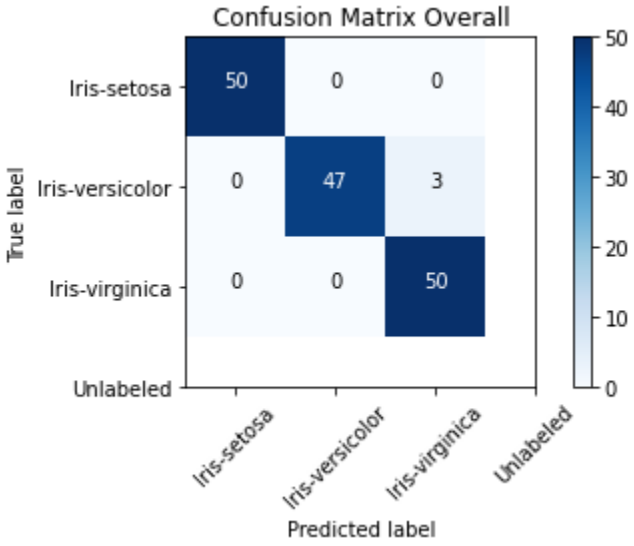

Figure 40 Overall confusion matrix for iris flower classification (100% training data distribution)

Based on the classifier visualizations shown before, it can be said that the classifier runs the risk of overfitting and not necessarily generalizing accurate rules, but rather odd ones, depending on the coverage of the input feature space.

Related work

Genetic algorithms

For the task of iris flower classification, another proposed solution that uses genetic algorithms comes from [11].
In this case, the iris flower dataset consisting of 150 records was split into 100 samples for the training data and 50 samples for the validation data, resulting in a roughly 66%-33% split.
In the following table, the results of the proposed method are compared with GAKPER, the method described in the mentioned work.

	Training accuracy			Validation accuracy		
	Correct	Incorrect	Unknown	Correct	Incorrect	Unknown
GAKPER	94%	0%	6%	91%	3%	6%
Proposed method	99%	1%	0%	98%	2%	0%

Table 2 Accuracy comparison between proposed method and another genetic algorithm

Although the results obtained are competitive according to the expectations set previously, preliminary conclusions are that the rules learnt by the classifier do not necessarily show potential for generalization to other real-life data, that being a matter of case-by-case analysis.

Support vector machines

A solution that comes under the shape of a support vector machine for iris flower classification comes from [8]. This work aims to shorten the time necessary for training such a machine by carefully selecting samples to remove from the training set, while still maintaining closely similar performances. The classifiers obtained are of one-versus-all nature, similar to what has been trained in this paper, and use a polynomial kernel. In this case, the dataset was split in halves between the training dataset and the validation dataset. For comparison, the solution obtained where no samples were removed from the training set, and with the best accuracy in the validation set, will be used.

	Validation accuracy		
	Iris setosa	Iris versicolor	Iris virginica
SVM (polynomial)	100%	97.33%	86.67%
Proposed method	100%	97.33%	97.33%

Table 3 Accuracy comparison between proposed method and support vector machines (1)

Other results from the applying of support vector machines to the same problem come from [10], which uses different several techniques, such as linear SVC, grid search tuned SVC and non-linear SVC. In this case, the data was split between training and validation data based on a 80%-20% ratio.

	Training accuracy	Validation accuracy
Linear SVC	97%	100%
On Tuning the C parameter	96%	100%
Non-Linear SVC	99.17%	100%
Proposed method	97.5%	96.66%

Table 4 Accuracy comparison between proposed method and support vector machines (2)

Artificial neural networks

Using an artificial neural network and particle swarm optimization, [9] has also pushed forward a

solution to iris flower classification. Unfortunately, in the case of this work, the ratio of data split between training and validation sets could not be identified. The results will be compared with the proposed method trained on all the samples from the dataset, although practicing cross-validation is recommended for the purpose of preventing overfit and obtaining a robust classifier.

	Accuracy
ANN + PSO	97.3%
Proposed method	98%

Table 5 Accuracy comparison between proposed method and neural network

Conclusion

Upon experimentation and comparison with other methods proposed by related work, the conclusion is that the proposed method is viable for classification, exhibiting non-linear character due to the underlying Perlin noise and yielding competitive results. However, it was observed that by having only accuracy as the only objective for the genetic algorithm that implements the learning, the resulting classifiers can either overfit the given data or learn rules that do not necessarily generalize the concepts underlined by the data. The proposed solutions are to introduce supplementary objectives that control the aspect of the underlying Perlin noise, and to inspect the visualizations of the produced classifiers in order to assess the existence of rather unpredictable patterns for the feature areas not covered by the input samples.

Further work

This paper focused on the classifier proposal and demonstration of its feasibility, but if more research is to be done in this direction, there are still aspects that are lacking in terms of coverage or which are not covered at all.

As such, a formal mathematical demonstration is required to assess whether there is a procedural noise that can accurately map to any problem solution space, or if there are limitations, and to identify them, regarding the capacities of procedural noise in modelling such a problem solution space. Such a demonstration could also come far enough as to explain whether this method could be applied not only to classification, but also to regression problems.

In terms of classifier evaluation, the time required for training and evaluation was not measured, and potential factors that could influence this metric are the quality of implementation and the use of time efficient, HPC methods to compute the value of the noise function in given points, such as GPU accelerated algorithms.

The implementation provided lacks usage of the hard mutation operator, but other attempts can make use of it by applying it with a low probability following crossover to expand the exploration aspect of the genetic algorithm

Bibliography

[1] H.-J. Bae, C.-W. Kim, N. Kim, B. Park, N. Kim, J.-B. Seo and S.-M. Lee, "A Perlin Noise-Based Augmentation Strategy for Deep Learning with Small Data Samples of HRCT Images," *Scientific Reports*, vol. 8, no. 17687, 2018.

[2] N. Inoue, E. Yamagata and H. Kataoka, "Initialization Using Perlin Noise for Training Networks with a Limited Amount of Data," in *2020 25th International Conference on Pattern Recognition (ICPR)*, 2021.

[3] K. T. Co, L. Muñoz-González, S. de Maupeou and E. C. Lupu, "Procedural noise adversarial examples for black-box attacks on deep convolutional networks," in *Proceedings of the 2019 ACM SIGSAC conference on computer and communications security*, 2019.

[4] R. Rombach, A. Blattmann, D. Lorenz, P. Esser and B. Ommer, "High-Resolution Image Synthesis with Latent Diffusion Models," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2022.

[5] J. C. Hart, "Perlin noise pixel shaders," in *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*, Los Angeles, 2001.

[6] R. A. Fisher, "The use of multiple measurements in taxonomic problems," *Annals of eugenics*, vol. 7, no. 2, pp. 179-188, 1936.

[7] K. Duan, S. S. Keerthi, W. Chu, S. K. Shevade and A. N. Poo, "Multi-category classification by soft-max combination of binary classifiers," in *International Workshop on Multiple Classifier Systems*, Heidelberg, 2003.

[8] R. Koggalage and S. Halgamuge, "Reducing the number of training samples for fast support vector machine classification," *Neural Information Processing-Letters and Review*, vol. 2, no. 3, pp. 57-65, 2004.

[9] A. Roy, D. Dutta and K. Choudhury, "Training artificial neural network using particle swarm optimization algorithm," *International Journal of Advanced Research in Computer Science and Software Engineering*, vol. 3, no. 3, pp. 430-434, 2013.

[10] A. Shukla, A. Agarwal, H. Pant and P. Mishra, "Flower classification using supervised learning," *International Journal of Engineering Research & Technology*, vol. 9, no. 5, pp. 757-762, 2020.

[11] J. Gopalan, R. Alhajj and K. Barker, "Discovering Accurate and Interesting Classification Rules Using Genetic Algorithm," in *Proceedings of the 2006 International Conference on Data Mining*, Las Vegas, 2006.

[12] K. Kira and L. A. Rendell, "A practical approach to feature selection," in *Machine learning proceedings*, 1992.

[13] A. Lagae, S. Lefebvre, R. L. Cook, T. Derose, G. Drettakis, D. S. Ebert, ... and M. Zwicker, "State of the Art in Procedural Noise Functions," Eurographics (State of the Art Reports), 2010.

[14] C. Tantithamthavorn, S. McIntosh, A. E. Hassan and K. Matsumoto, "The impact of automated parameter optimization on defect prediction models," *IEEE Transactions on Software Engineering*, vol. 45, no. 7, pp. 683-711, 2018.

[15] I. Ilhan and G. Tezel, "A genetic algorithm–support vector machine method with parameter optimization for selecting the tag SNPs," *Journal of biomedical informatics*, vol. 46, no. 2, pp. 328-340, 2013.

[16] D. Baehrens, T. Schroeter, S. Harmeling, M. Kawanabe, K. Hansen and K. R. Müller, "How to explain individual classification decisions," *The Journal of Machine Learning Research*, vol. 11,

pp. 1803-1831, 2010.

[17] C. Cortes and V. Vapnik, "Support-vector networks," *Machine learning*, vol. 20, no. 3, pp. 273-297, 1995.

[18] M. Marvin and A. P. Seymour, "Perceptrons," *MA: MIT Press*, no. 6, pp. 318-362, 1969.

Table of figures

Figure 1 An example of loss values in a solution space (Source: https://umu.to/blog/2018/06/29/hill-climbing-irl) 3
Figure 2 An example of terrain mesh generated using Perlin noise (Source: https://www.scratchapixel.com/lessons/procedural-generation-virtual-worlds/perlin-noise-part-2/perlin-noise-terrain-mesh) 3
Figure 3 Example of scaling across a dimension in a Perlin noise 4
Figure 4 Changing the number of octaves or manipulating turbulence in a Perlin noise 4
Figure 5 Manipulating the threshold applied to the same Perlin noise by changing brightness value 5
Figure 6 Evolution graph of mean and best fitness in populations across generations in AND logic gate learning (x = generation, y = fitness) 9
Figure 7 Obtained AND classifier and original Perlin noise graph 9
Figure 8 Evolution graph of mean and best fitness in populations across generations in OR logic gate learning (x = generation, y = fitness) 10
Figure 9 Obtained OR classifier and original Perlin noise graph 10
Figure 10 Evolution graph of mean and best fitness in populations across generations in XOR logic gate learning (x = generation, y = fitness) 11
Figure 11 Obtained XOR classifier and original Perlin noise graph 11
Figure 12 Distribution of data in the iris flower dataset 12
Figure 13 Evolution graph in Iris setosa classifier learning (50% training data distribution) 13
Figure 14 Obtained Iris setosa classifier and original Perlin noise graph (50% training data distribution) 14
Figure 15 Evolution graph in Iris versicolor classifier learning (50% training data distribution) 14
Figure 16 Obtained Iris versicolor classifier and original Perlin noise graph (50% training data distribution) 15
Figure 17 Evolution graph in Iris virginica classifier learning (50% training data distribution) 15
Figure 18 Obtained Iris virginica classifier and original Perlin noise graph (50% training data distribution) 16
Figure 19 Training, validation and overall confusion matrices for iris flower classification (50% training data distribution) 17
Figure 20 Evolution graph in Iris setosa classifier learning (66% training data distribution) 17
Figure 21 Obtained Iris setosa classifier and original Perlin noise graph (66% training data distribution) 18
Figure 22 Evolution graph in Iris versicolor classifier learning (66% training data distribution) 18
Figure 23 Obtained Iris versicolor classifier and original Perlin noise graph (66% training data distribution) 19
Figure 24 Evolution graph in Iris virginica classifier learning (66% training data distribution) 19
Figure 25 Obtained Iris virginica classifier and original Perlin noise graph (66% training data distribution) 20
Figure 26 Training, validation and overall confusion matrices for iris flower classification (66%

training data distribution) 21
Figure 27 Evolution graph in Iris setosa classifier learning (80% training data distribution) 21
Figure 28 Obtained Iris setosa classifier and original Perlin noise graph (80% training data distribution) 22
Figure 29 Evolution graph in Iris versicolor classifier learning (80% training data distribution) 22
Figure 30 Obtained Iris versicolor classifier and original Perlin noise graph (80% training data distribution) 23
Figure 31 Evolution graph in Iris virginica classifier learning (80% training data distribution) 23
Figure 32 Obtained Iris virginica classifier and original Perlin noise graph (80% training data distribution) 24
Figure 33 Training, validation and overall confusion matrices for iris flower classification (80% training data distribution) 25
Figure 34 Evolution graph in Iris setosa classifier learning (100% training data distribution) 25
Figure 35 Obtained Iris setosa classifier and original Perlin noise graph (100% training data distribution) 26
Figure 36 Evolution graph in Iris versicolor classifier learning (100% training data distribution) 26
Figure 37 Obtained Iris versicolor classifier and original Perlin noise graph (100% training data distribution) 27
Figure 38 Evolution graph in Iris virginica classifier learning (100% training data distribution) 27
Figure 39 Obtained Iris virginica classifier and original Perlin noise graph (100% training data distribution) 28
Figure 40 Overall confusion matrix for iris flower classification (100% training data distribution) 28

Table 1 Truth values for AND, OR and XOR logic gates 8
Table 2 Accuracy comparison between proposed method and another genetic algorithm 29
Table 3 Accuracy comparison between proposed method and support vector machines (1) 29
Table 4 Accuracy comparison between proposed method and support vector machines (2) 29
Table 5 Accuracy comparison between proposed method and neural network 30