Statistical analysis of experimental data
Machine Learning

Aleksander Filip Żarnecki

FACULTY OF PHYSICS
UNIVERSITY OF WARSAW

Lecture 13

January 16, 2025

Machine Learning

Problem definition

The problem is similar to the one discussed in lecture 10: we want to discriminate between two model hypothesis H_0 and H_1 based on the collected data D.

Different case - **classification of collected measurements**:

- H_0 - measurement can be attributed to the Standard Model (background),
- H_1 - measurement is due to BSM contribution (signal),
- D - single measurement ("event" in HEP experiments)

According to Neymann and Pearson, the optimal, "most powerful" method to discriminate between the two hypothesis is to look at likelihood ratio

$$Q(D) = \frac{L(D|H_1)}{L(D|H_0)}$$

Classification errors

O.Behnke et. al, *Data Analysis in High Enegy Physics*

Selecting the classification criteria (cut), two types of error need to be considered

	Reject H_0 (select as signal)	**Accept H_0 (select as background)**
H_0 is false (event is signal)	Right decision with probability $1-\beta$ = power = efficiency	Wrong decision; type II error with probability β
H_0 is true (event is background)	Wrong decision; type I error with probability α = size = significance	Right decision with probability $1-\alpha$ = background rejection

Probability of accepting fake

$$\alpha = \int_{\text{accepted}} dx\ p(x|H_0)$$

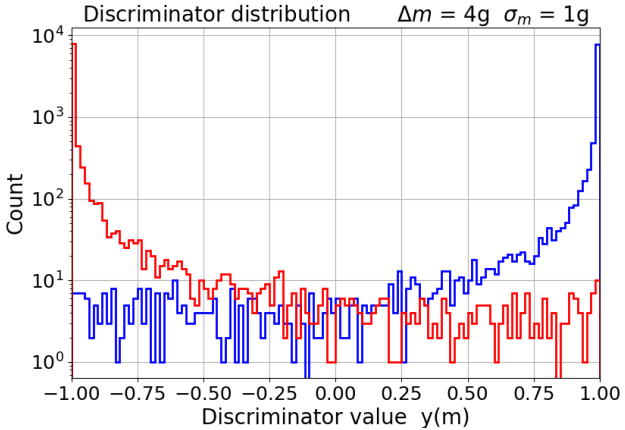
Probability of rejecting good

$$\beta = \int_{\text{rejected}} dx\ p(x|H_1)$$

Simple example

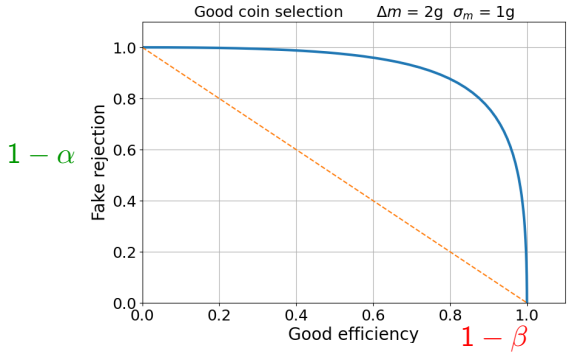
Discriminator function distribution

expect $y \to -1$ for fake coin, $y \to +1$ for good coin



ROC curve

For both good and fake coins, efficiency depends on the assumed y_{cut} value.
All possible choices on a Receiver-Operating-Characteristic curve:



In the realistic case, we can not have $\alpha \to 0$ and $\beta \to 0$ at the same time...
Optimal cut value strongly depends on the actual goal of the analysis...

Linear discriminant (Behnke)

Classifier based on the linear combination of input variables:

$$F(\mathbf{x};\mathbf{w}) \;=\; w_0 \;+\; \sum_{j=1}^{N} w_j\, x_j \;=\; w_0 \;+\; \mathbf{w}\cdot\mathbf{x}$$

Resulting decision boundaries, $F(\mathbf{x}) = F_{cut}$, are hyperplanes in N dim.

Weight vector $\mathbf{w}$ defines the direction, on which all events are projected.
Projection "reduces" the N variable problem to single variable $F(\mathbf{x})$.

If we assume Gaussian variable distributions, we can look at the direction which maximizes the relative distance between the two hypothesis in F:

$$D(\mathbf{w}) \;=\; \frac{(h_1 - h_0)^2}{\sigma_1^2 + \sigma_0^2}$$

h_k and σ_k^2 are the expected values and variances of $F(\mathbf{x})$ for hypothesis k.

Linear discriminant (Behnke)

Classifier based on the linear combination of input variables:

$$F(\mathbf{x};\mathbf{w}) \;=\; w_0 \;+\; \sum_{j=1}^{N} w_j\, x_j \;=\; w_0 \;+\; \mathbf{w}\cdot\mathbf{x}$$

Resulting decision boundaries, $F(\mathbf{x}) = F_{cut}$, are hyperplanes in N dim.

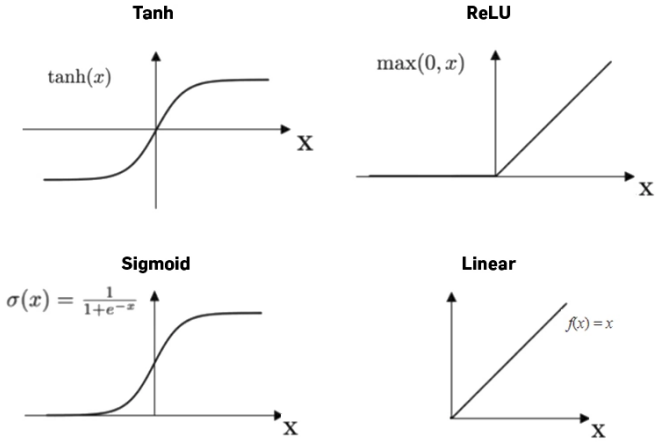
Weight vector $\mathbf{w}$ defines the direction, on which all events are projected.
Projection "reduces" the N variable problem to single variable $F(\mathbf{x})$.

However, the problem can be also solved without looking at the global properties, by minimizing the "loss function". Possible choice, "distance":

$$L(\mathbf{w}) \;=\; \sum_{\text{events } i} \Big[t^{(i)} - y(F(\mathbf{x}^{(i)};\mathbf{w})) \Big]^2$$

where y is the "activation function", $t^{(i)}$ is the true class of event $\mathbf{x}^{(i)}$.

Activation function



Source: Artificial Intelligence Wiki

Perceptron Learning **"Learning on errors"**

One can consider the iterative procedure of adjusting the weights:

$$\mathbf{w}^{(n+1)} = \mathbf{w}^{(n)} - \eta \sum_i \left(y^{(i)} - t^{(i)} \right) \cdot \mathbf{x}^{(i)}$$

where η is the learning rate parameter.

Events which are incorrectly classified contribute most to loss function.
They also have largest impact in the weigh adjustment procedure...

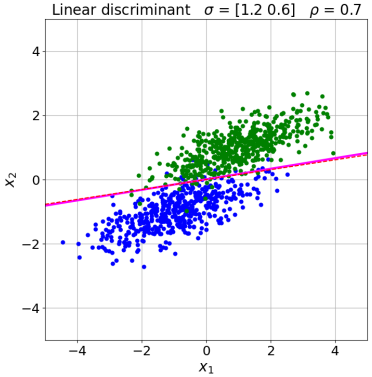
This approach was first proposed by M. Rosenblatt in 1958.

Weight correction can be applied on event by event basis (starting from the beginning when event loop completed) or calculating global correction for the whole sample.

Surprisingly, with proper choice of η this procedure works, results in classification optimization, even without referring to the loss function...

Perceptron Learning example

Example results for linear discriminant, starting from random weights:

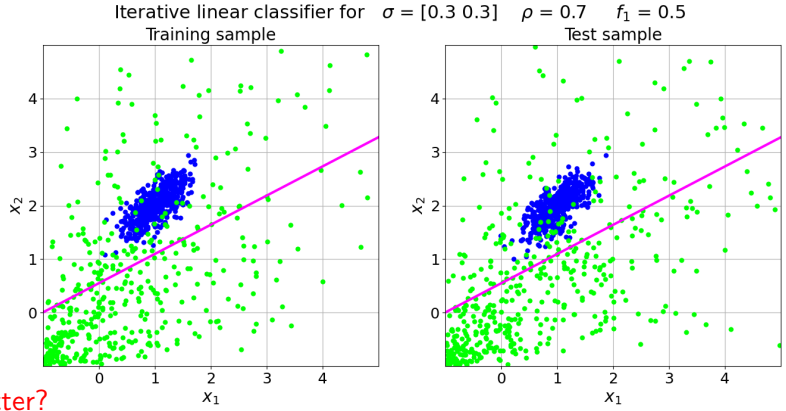


Iterative procedure (solid magenta) compared with Fisher discriminant (dashed red)

Machine Learning

Linear discriminant Single percepton training

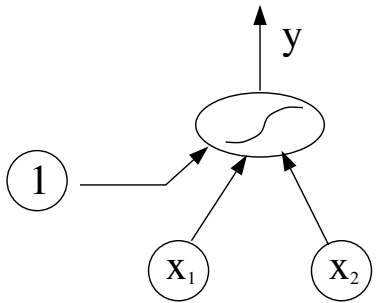
Linear discriminant is quite effective for separation of two Gaussian samples, but clearly not optimal for more complicated cases



Can we do better?

Single percepton

We can present the data flow in as a simple diagram:



Classification is based on the output y of the activation function.

Activation function is calculated for a linear combination of three inputs:

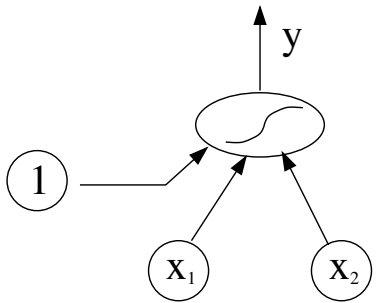
- two input variables, x_1 and x_2
- constant offset (1)

Input weights can be found in the iterative "learning procedure"

$$\mathbf{w}^{(n+1)} = \mathbf{w}^{(n)} - \eta \sum_{events} \left(y^{(i)} - t^{(i)} \right) \cdot \mathbf{x}^{(i)}$$

Single percepton

We can present the data flow in as a simple diagram:



Classification is based on the output y of the activation function.

Activation function is calculated for a linear combination of three inputs:

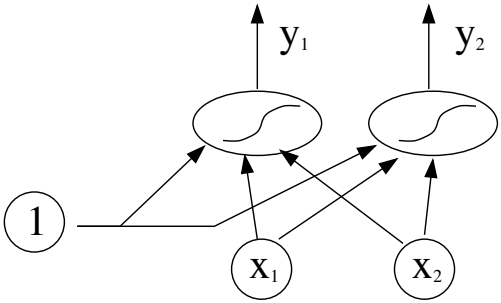
- two input variables, x_1 and x_2
- constant offset (1)

Input weights can be found in the iterative "learning procedure"

But single linear combination always results in a linear decision boundary...

Two perceptons

We can try to train two independent classifiers:



If starting from random initial weights, training results could be different...

But how to combine them?

Two percepton layers

It seems quite natural to add additional percepton to combine the two...

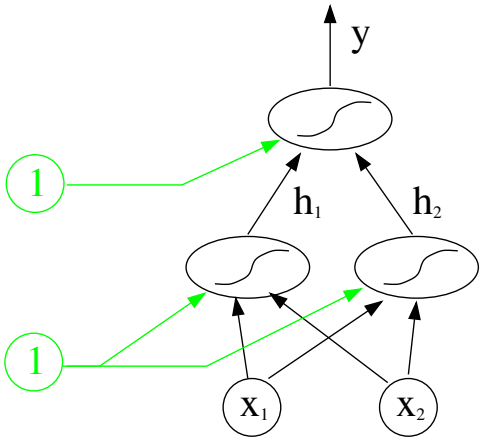
Output-layer neuron:

$$y = f\left(w_0^{(1)} + \sum_{j=1}^{2} w_j^{(1)} h_j\right)$$

Hidden-layer neuron:

$$h_j = f\left(w_{j,0}^{(2)} + \sum_{k=1}^{2} w_{j,k}^{(2)} x_k\right)$$

$\Rightarrow$ nine independent weights
one for each arrow



Learning rules

Miroslav Kubat, *An Introduction to Machine Learning*

Backpropagation of Errors: contribution of event i to the weight-adjusting procedure is proportional to the classification error:

$$\delta_i^{(1)} = (y_i - t_i)$$

Learning rules

Miroslav Kubat, *An Introduction to Machine Learning*

Backpropagation of Errors: contribution of event i to the weight-adjusting procedure is proportional to the classification error:

$$\delta_i^{(1)} = (y_i - t_i)\,(1 - y_i)\,(1 + y_i)$$

Additional factor reduces impact of "well classified" events, $y \to \pm 1$
$\Rightarrow$ we focus on those for which classification was "weak", $y_i \sim 0$.

Learning rules

Miroslav Kubat, *An Introduction to Machine Learning*

Backpropagation of Errors: contribution of event i to the weight-adjusting procedure is proportional to the classification error:

$$\delta_i^{(1)} = (y_i - t_i)\,(1 - y_i)\,(1 + y_i)$$

Additional factor reduces impact of "well classified" events, $y \to \pm 1$
$\Rightarrow$ we focus on those for which classification was "weak", $y_i \sim 0$.

For the output layer neurons, we can apply procedure similar to the percepton learning:

$$\mathbf{w}^{(1)(n+1)} = \mathbf{w}^{(1)(n)} - \eta \sum_i \delta_i^{(1)} \cdot \mathbf{h}_i$$

where $\mathbf{h}_i$ is the vector of hidden layer results + offset

Learning rules

For hidden layer, we need to define the corresponding "error" for each node j.
We "back propagate" it for each event from the output node:

$$\delta_{j,i}^{(2)} = w_j^{(1)} \delta_i^{(1)} (1 - h_{j,i}) (1 + h_{j,i})$$

where we include weight $w_j^{(1)}$ connecting given node to output neuron.
Again, we suppress impact of events with "strong opinion".

Learning rules

For hidden layer, we need to define the corresponding "error" for each node j.
We "back propagate" it for each event from the output node:

$$\delta_{j,i}^{(2)} = w_j^{(1)} \delta_i^{(1)} (1 - h_{j,i}) (1 + h_{j,i})$$

where we include weight $w_j^{(1)}$ connecting given node to output neuron.
Again, we suppress impact of events with "strong opinion".

Weight update rule for hidden layer neurons:

$$\mathbf{w}_j^{(2)(n+1)} = \mathbf{w}_j^{(2)(n)} - \eta \sum_i \delta_{j,i}^{(2)} \cdot \mathbf{x}_i$$

Learning rules

For hidden layer, we need to define the corresponding "error" for each node j.
We "back propagate" it for each event from the output node:

$$\delta_{j,i}^{(2)} = w_j^{(1)}\, \delta_i^{(1)}\, (1 - h_{j,i})\, (1 + h_{j,i})$$

where we include weight $w_j^{(1)}$ connecting given node to output neuron.
Again, we suppress impact of events with "strong opinion".

Weight update rule for hidden layer neurons:

$$\mathbf{w}_j^{(2)(n+1)} = \mathbf{w}_j^{(2)(n)} - \eta \sum_i \delta_{j,i}^{(2)} \cdot \mathbf{x}_i$$

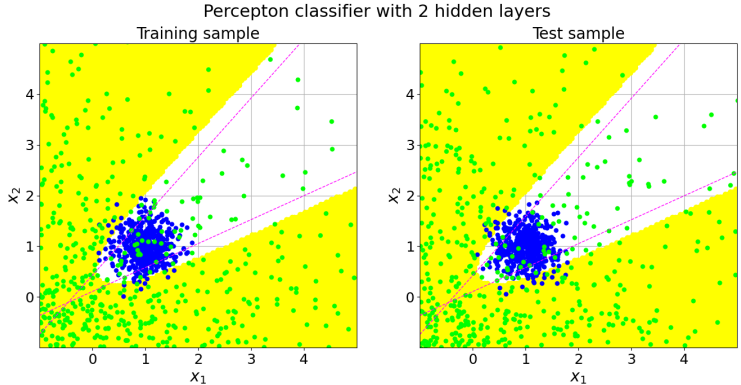
Iterative procedure, starting from random weights:

- calculate y_i for train sample events $\Rightarrow$ extract $\delta_i^{(1)}$ and $\delta_{j,i}^{(2)}$
- update $\mathbf{w}^{(1)}$ and $\mathbf{w}_j^{(2)}$, decrease η, repeat from the beginning

Simplest case

13_NN.ipynb Open in Colab

Simplest network: one hidden layer with two preceptons...
Visible improvement in efficiency and flexibility of classification!



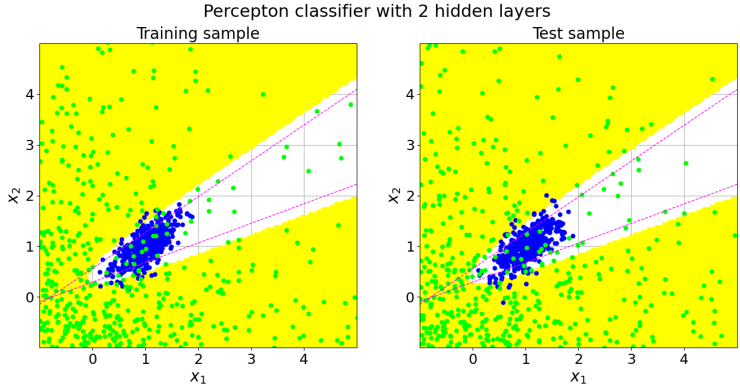
Correlation for signal sample $\rho = 0$

Simplest case

13_NN.ipynb Open in Colab

Simplest network: one hidden layer with two preceptons...

Visible improvement in efficiency and flexibility of classification!

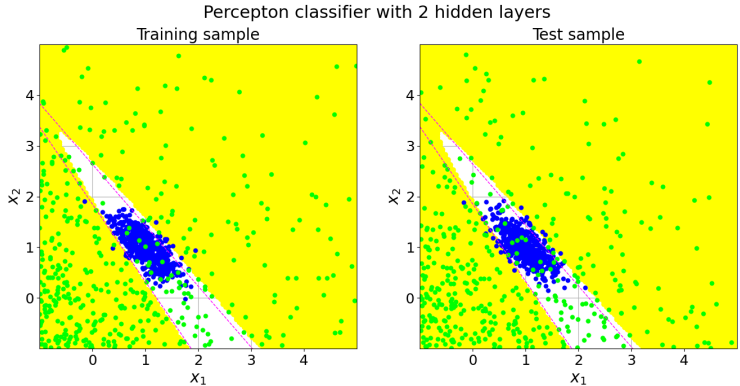


Correlation for signal sample $\rho = 0.7$

Simplest case

13_NN.ipynb Open in Colab

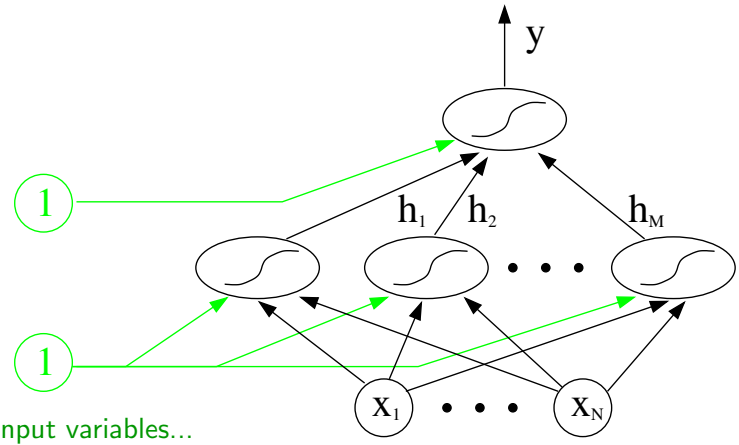
Simplest network: one hidden layer with two preceptons...
Visible improvement in efficiency and flexibility of classification!



Correlation for signal sample $\rho = -0.7$

More complex case

We can have arbitrary number of neurons in hidden layer...



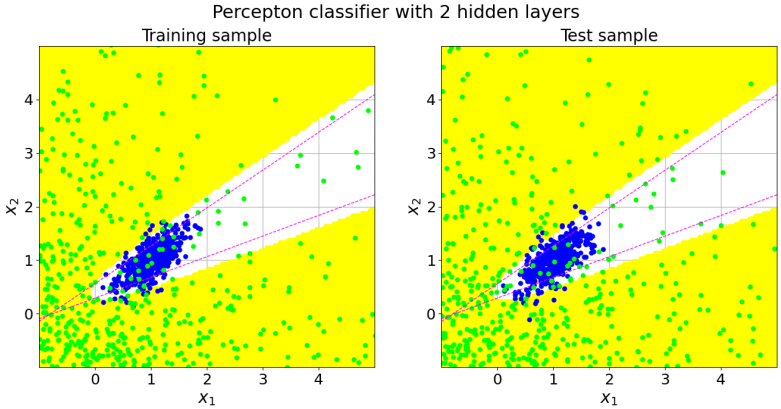
as well as more input variables...

More complex case

13_NN.ipynb Open in Colab

Classification improves with the number of nodes in the hidden layer.

Learning takes a little bit longer, but we can gain a lot...

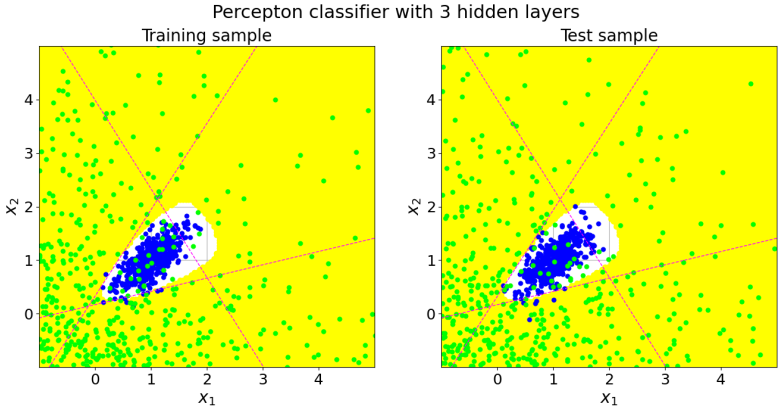


More complex case

13_NN.ipynb Open in Colab

Classification improves with the number of nodes in the hidden layer.

Learning takes a little bit longer, but we can gain a lot...

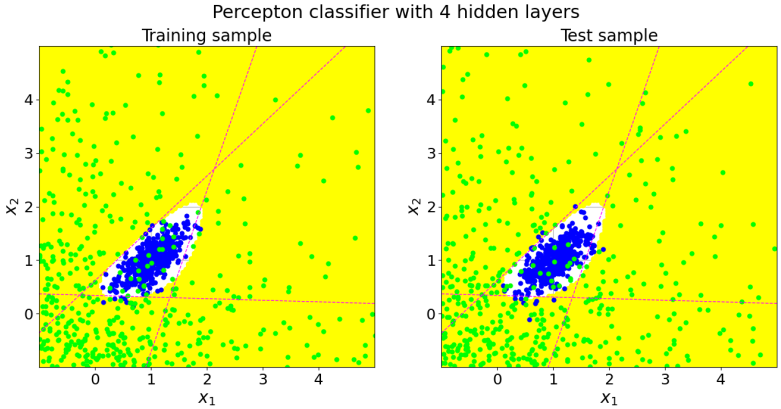


More complex case

13_NN.ipynb Open in Colab

Classification improves with the number of nodes in the hidden layer.

Learning takes a little bit longer, but we can gain a lot...

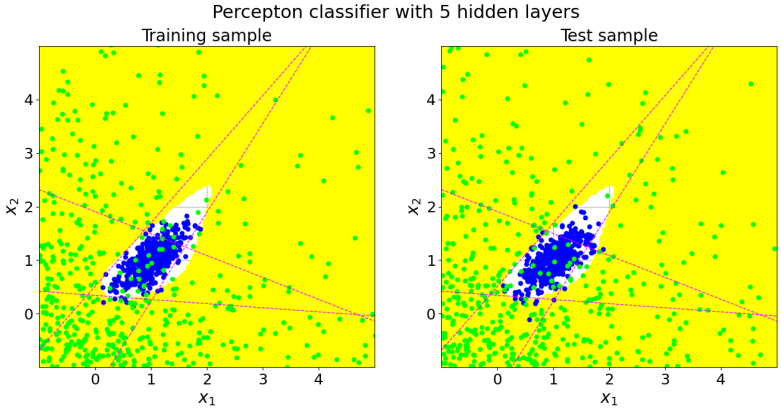


More complex case

13_NN.ipynb Open in Colab

Classification improves with the number of nodes in the hidden layer.

Learning takes a little bit longer, but we can gain a lot...

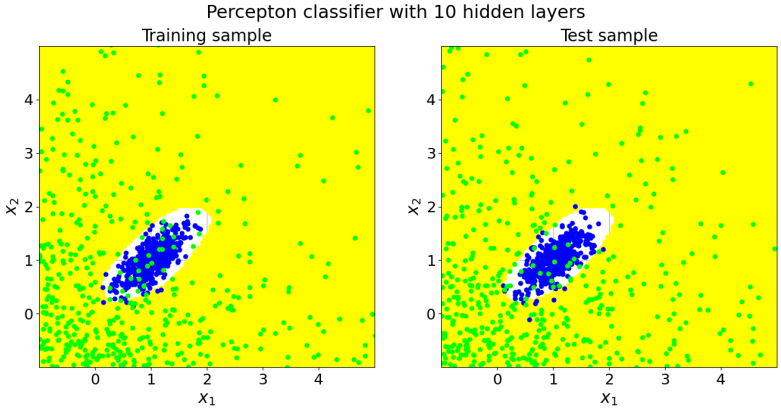


More complex case

13_NN.ipynb Open in Colab

Classification improves with the number of nodes in the hidden layer.

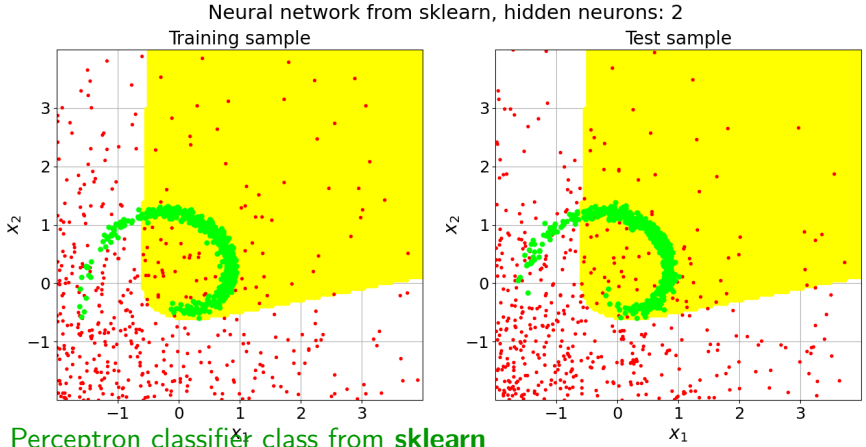
Learning takes a little bit longer, but we can gain a lot... for simple distribution...



Multilayer example

13_NNsk.ipynb Open in Colab

One hidden layer only - limited shape flexibility



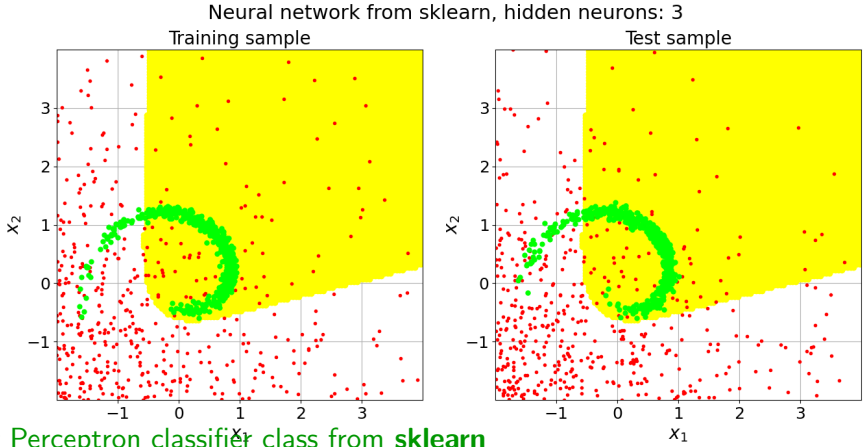
Multi-layer Perceptron classifier class from **sklearn**

Multilayer example

13_NNsk.ipynb

Open in Colab

One hidden layer only - limited shape flexibility



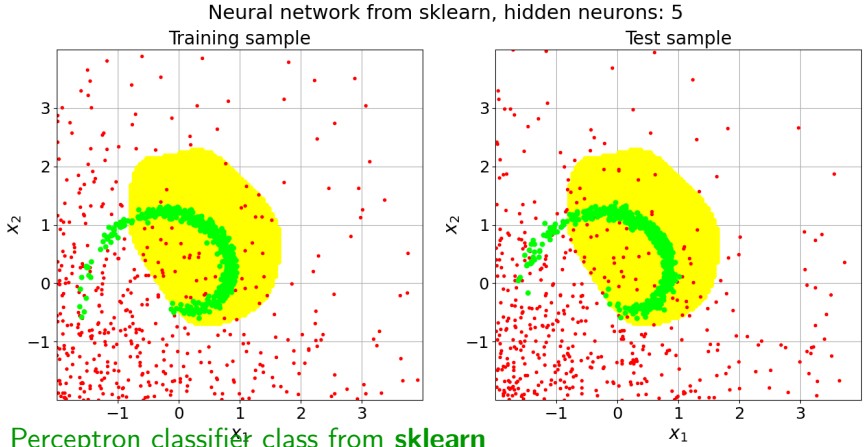
Multi-layer Perceptron classifier class from **sklearn**

Multilayer example

13_NNsk.ipynb

Open in Colab

One hidden layer only - limited shape flexibility



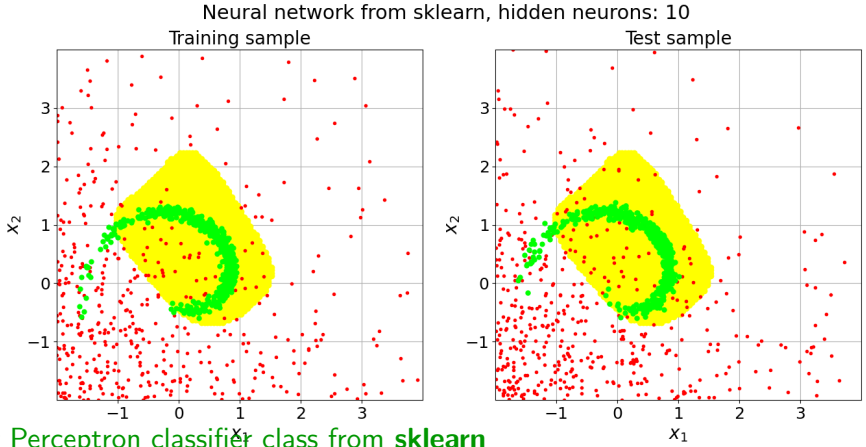
Multi-layer Perceptron classifier class from **sklearn**

Multilayer example

13_NNsk.ipynb

Open in Colab

One hidden layer only - limited shape flexibility



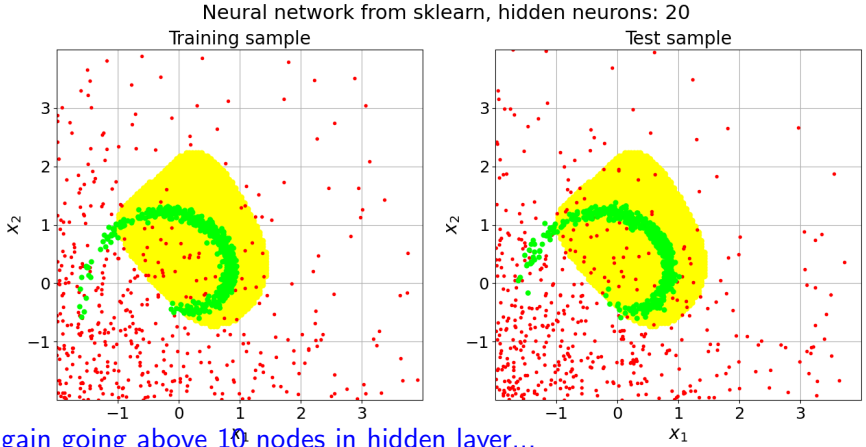
Multi-layer Perceptron classifier class from **sklearn**

Multilayer example

13_NNsk.ipynb

Open in Colab

One hidden layer only - limited shape flexibility



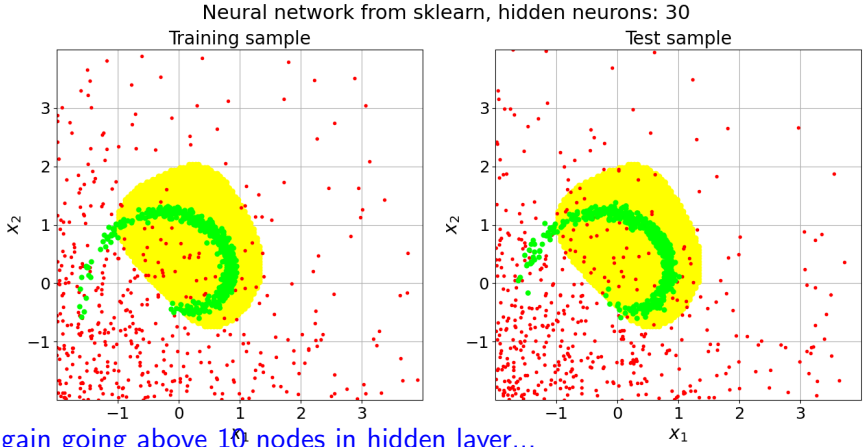
Not much gain going above 10 nodes in hidden layer...

Multilayer example

13_NNsk.ipynb

Open in Colab

One hidden layer only - limited shape flexibility



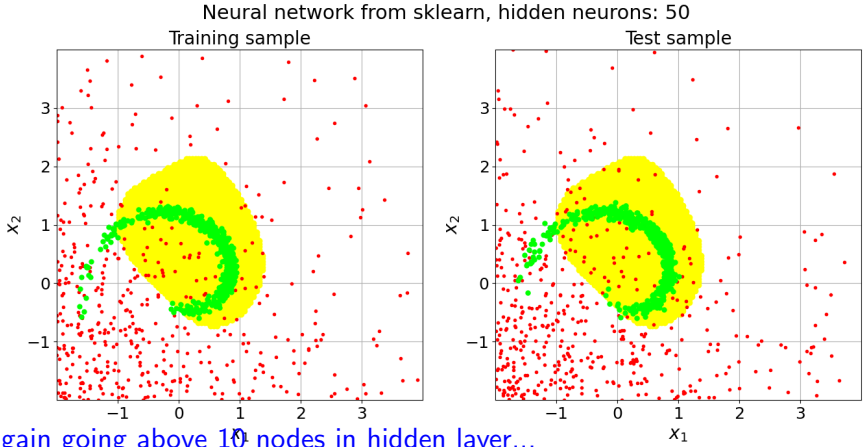
Not much gain going above 10 nodes in hidden layer...

Multilayer example

13_NNsk.ipynb

Open in Colab

One hidden layer only - limited shape flexibility



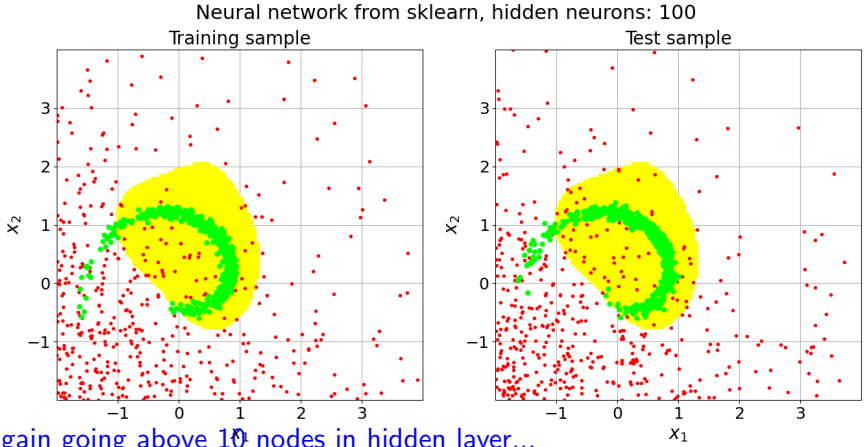
Not much gain going above 10 nodes in hidden layer...

Multilayer example

13_NNsk.ipynb

Open in Colab

One hidden layer only - limited shape flexibility



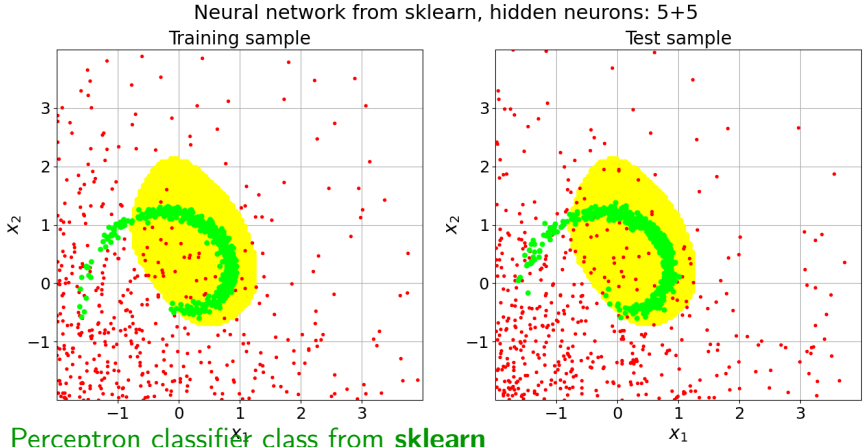
Not much gain going above 10 nodes in hidden layer...

Multilayer example

13_NNsk.ipynb

Open in Colab

Two hidden layer - more shape flexibility



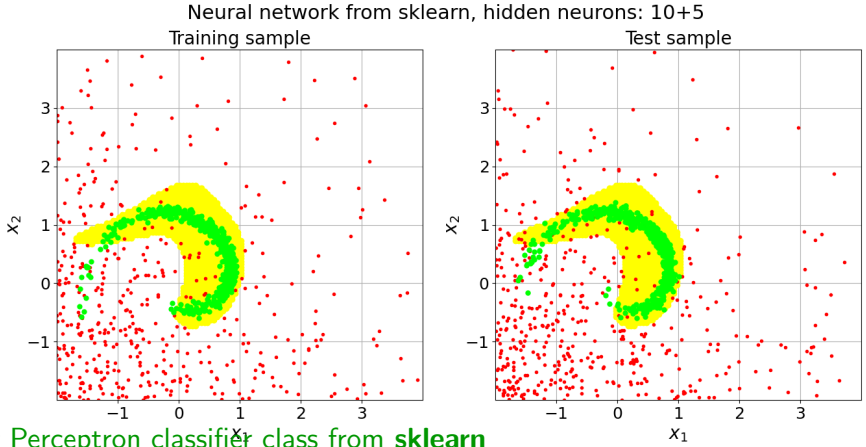
Multi-layer Perceptron classifier class from **sklearn**

Multilayer example

13_NNsk.ipynb

Open in Colab

Two hidden layer - more shape flexibility



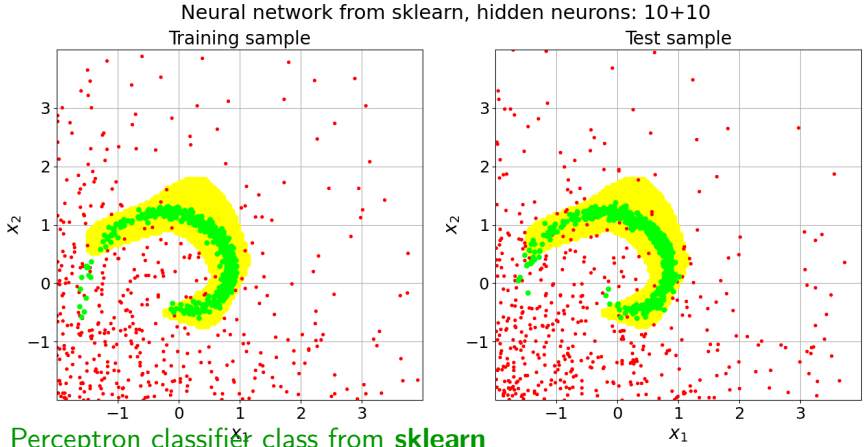
Multi-layer Perceptron classifier class from **sklearn**

Multilayer example

13_NNsk.ipynb

Open in Colab

Two hidden layer - more shape flexibility



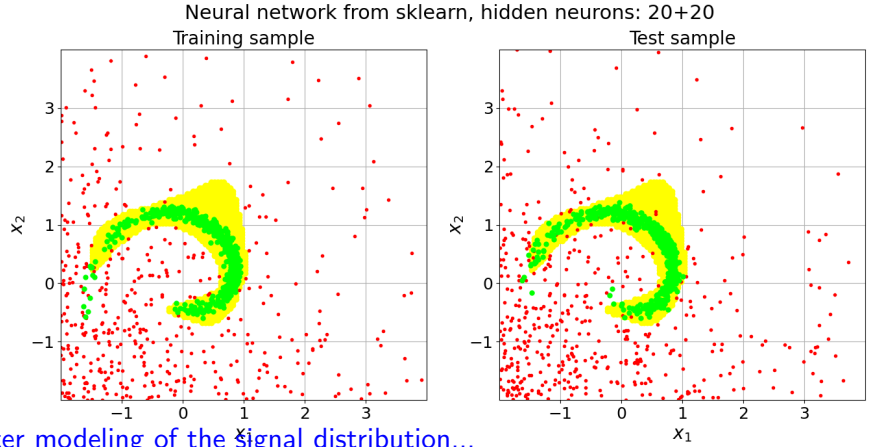
Multi-layer Perceptron classifier class from **sklearn**

Multilayer example

13_NNsk.ipynb

Open in Colab

Two hidden layer - more shape flexibility



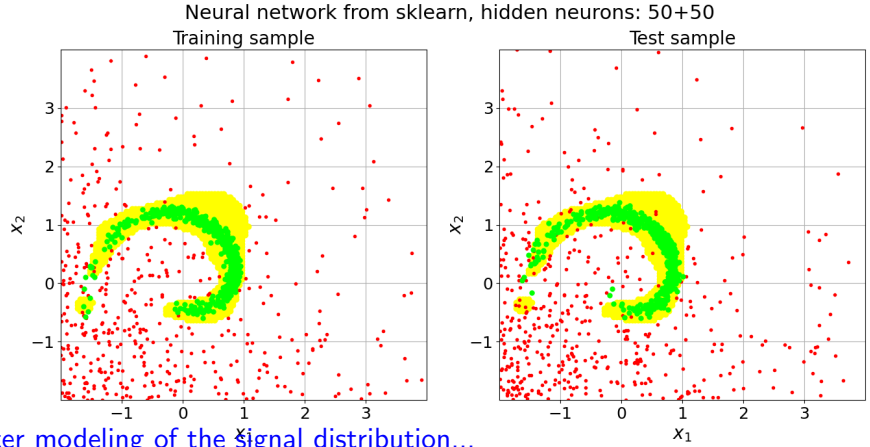
Much better modeling of the signal distribution...

Multilayer example

13_NNsk.ipynb

Open in Colab

Two hidden layer - more shape flexibility

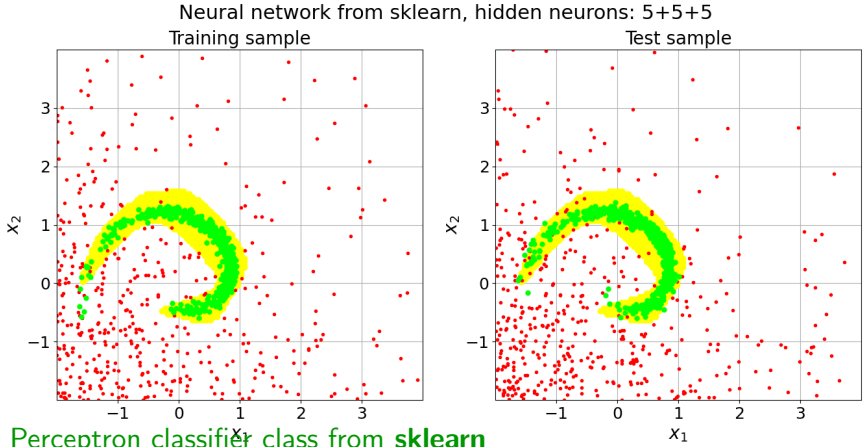


Much better modeling of the signal distribution...

Multilayer example

13_NNsk.ipynb Open in Colab

Three hidden layer - more details can be included

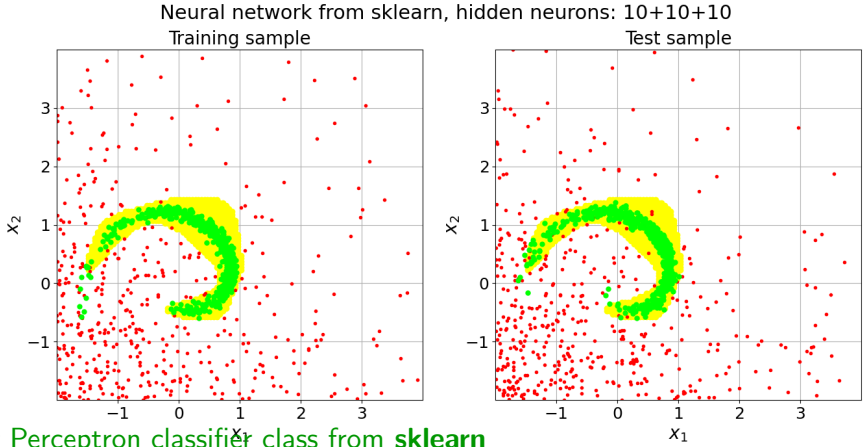


Multi-layer Perceptron classifier class from **sklearn**

Multilayer example

13_NNsk.ipynb Open in Colab

Three hidden layer - more details can be included

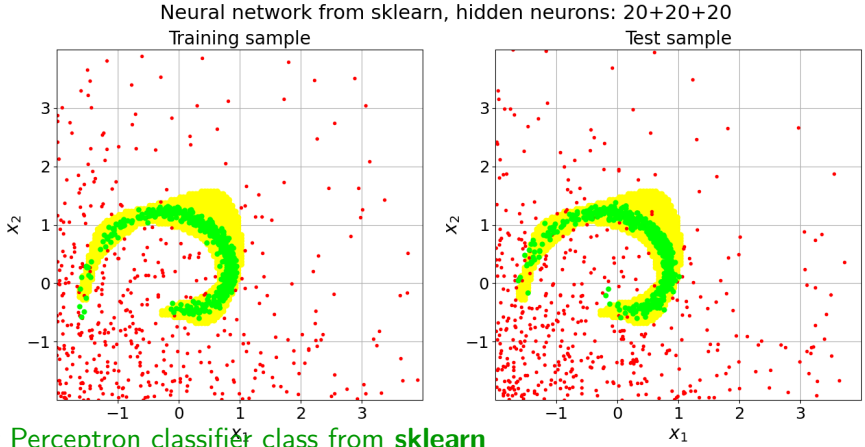


Multi-layer Perceptron classifier class from **sklearn**

Multilayer example

13_NNsk.ipynb Open in Colab

Three hidden layer - more details can be included

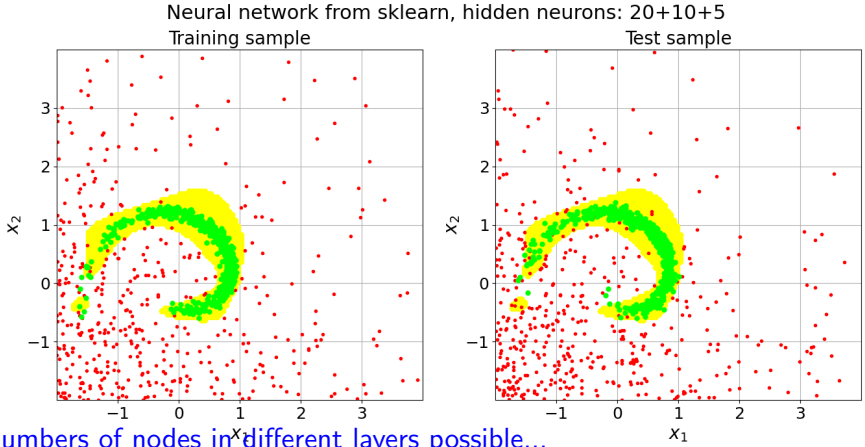


Multi-layer Perceptron classifier class from **sklearn**

Multilayer example

13_NNsk.ipynb Open in Colab

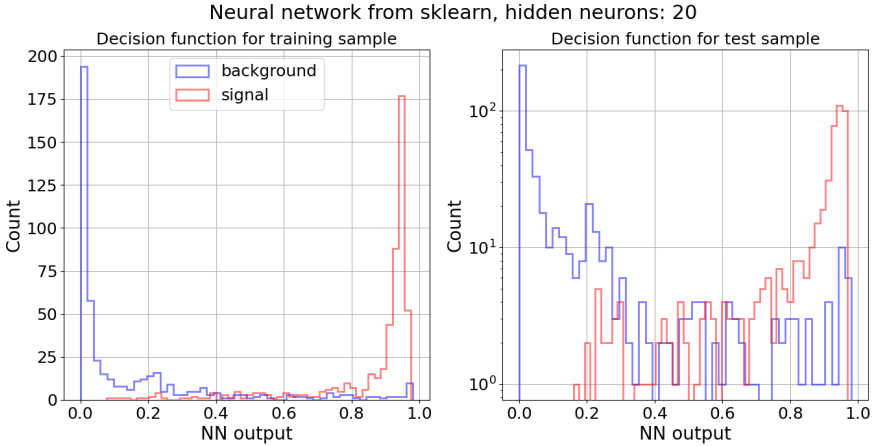
Three hidden layer - more details can be included



Different numbers of nodes in different layers possible...

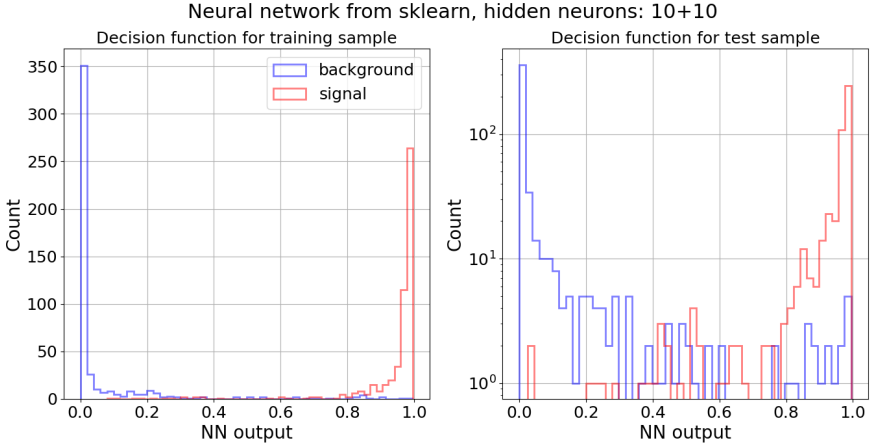
Comparison of the output discriminator function distribution

Single hidden layer with 20 neurons



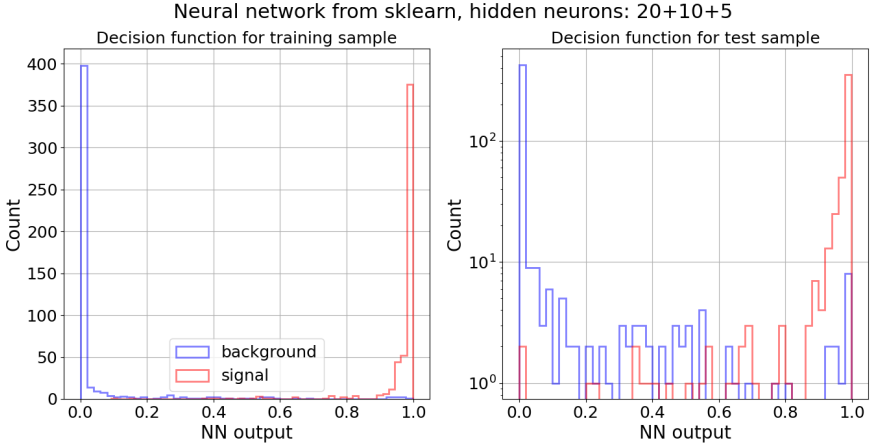
Comparison of the output discriminator function distribution

Two hidden layers, with 10 neurons each



Comparison of the output discriminator function distribution

Three hidden layers, with 20, 5 and 2 neurons clear improvement of the event classification



sklearn tips... https://scikit-learn.org/

Multi-layer perceptron is sensitive to variable scales.

It is highly recommended to scale input data, so each variable has the same range (eg. $[-1, +1]$) or same mean and variance (eg. $\mu = 0$ and $\sigma = 1$).
Both training and test samples need to be scaled in the same way!

sklearn tips... https://scikit-learn.org/

Multi-layer perceptron is sensitive to variable scales.

It is highly recommended to scale input data, so each variable has the same range (eg. $[-1, +1]$) or same mean and variance (eg. $\mu = 0$ and $\sigma = 1$).
Both training and test samples need to be scaled in the same way!

Different, more advanced learning algorithms are implemented in **sklearn**, one can choose between them with 'solver' parameter.

- 'lbfgs' converges faster and with better solutions on small datasets.
- For relatively large datasets, 'adam' is very robust.
  It usually converges quickly and gives pretty good performance.
- 'sgd' can perform best if learning rate is correctly tuned.

Machine Learning

Ensemble methods

It is relatively easy (in most cases) to design a classification algorithm which will result in the classification efficiency (fraction of correct classifications) slightly above 50% (random classification level).

Such classifiers are called "weak classifiers"

Ensemble methods

It is relatively easy (in most cases) to design a classification algorithm which will result in the classification efficiency (fraction of correct classifications) slightly above 50% (random classification level).

Such classifiers are called "weak classifiers"

It is much more difficult (in most realistic cases) to design a single classifier, which will result in efficiency close to 100% (error-less classification).

Such classifiers are called "strong classifiers"

Ensemble methods

It is relatively easy (in most cases) to design a classification algorithm which will result in the classification efficiency (fraction of correct classifications) slightly above 50% (random classification level).

Such classifiers are called "weak classifiers"

It is much more difficult (in most realistic cases) to design a single classifier, which will result in efficiency close to 100% (error-less classification).

Such classifiers are called "strong classifiers"

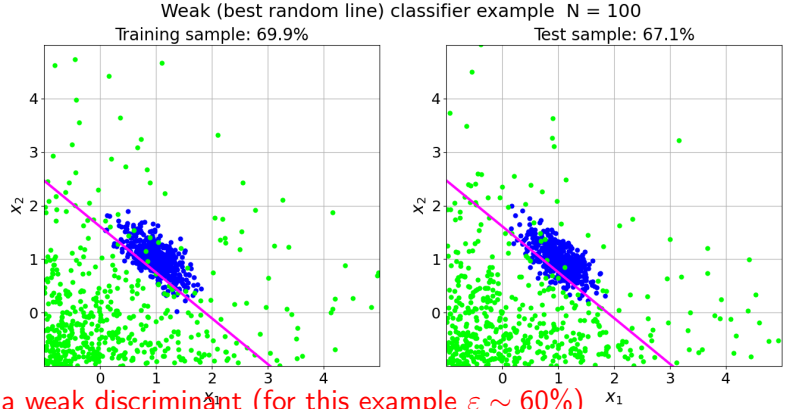
However, it turns out that one can build a strong classifier from many weak classifiers!

This is the underlying principle in many machine learning techniques...

Example weak discriminant

13_Weak.ipynb Open in Colab

Generate $N_{try} = 100$ random linear discriminants. Select the one with the highest efficiency (highest number of properly classified events).



This is clearly a weak discriminant (for this example $\varepsilon \sim 60\%$)

Ensemble methods

https://scikit-learn.org/

Two families of ensemble methods are usually distinguished:

- In averaging methods, the driving principle is to build several estimators independently and then to average their predictions.
  On average, the combined estimator is usually better than any of the single base estimator because its variance is reduced.

Ensemble methods

https://scikit-learn.org/

Two families of ensemble methods are usually distinguished:

- In averaging methods, the driving principle is to build several estimators independently and then to average their predictions.
  On average, the combined estimator is usually better than any of the single base estimator because its variance is reduced.
- In boosting methods, base estimators are built sequentially and one tries to reduce the bias of the combined estimator. The motivation is to combine several weak models to produce a powerful ensemble.

Ensemble methods

https://scikit-learn.org/

Two families of ensemble methods are usually distinguished:

- In averaging methods, the driving principle is to build several estimators independently and then to average their predictions.
  On average, the combined estimator is usually better than any of the single base estimator because its variance is reduced.
- In boosting methods, base estimators are built sequentially and one tries to reduce the bias of the combined estimator. The motivation is to combine several weak models to produce a powerful ensemble.

The two methods can also be combined...

Procedure

Let as assume that we have a sample of events $\mathbf{x}_i$ with true categories t_i.
All events have the same initial weight $w_i^{(1)} = 1/N$

Procedure

Let as assume that we have a sample of events $\mathbf{x}_i$ with true categories t_i.
All events have the same initial weight $w_i^{(1)} = 1/N$

The iterative procedure looks like follows. In step j:

1. train classifier C_j using our input data $\mathbf{x}_i$ with weights $w_i^{(j)}$

Procedure

Let as assume that we have a sample of events $\mathbf{x}_i$ with true categories t_i.
All events have the same initial weight $w_i^{(1)} = 1/N$

The iterative procedure looks like follows. In step j:

1. train classifier C_j using our input data $\mathbf{x}_i$ with weights $w_i^{(j)}$
2. calculate classifier response: $y_i^{(j)} = C_j(\mathbf{x}_i)$

Procedure

Let as assume that we have a sample of events $\mathbf{x}_i$ with true categories t_i.
All events have the same initial weight $w_i^{(1)} = 1/N$

The iterative procedure looks like follows. In step j:

1. train classifier C_j using our input data $\mathbf{x}_i$ with weights $w_i^{(j)}$
2. calculate classifier response: $y_i^{(j)} = C_j(\mathbf{x}_i)$
3. calculate classifier error rate: $\varepsilon_j = \sum w_i^{(j)} \cdot (y_i^{(j)} != t_i) \; / \sum w_i^{(j)}$

Procedure

Let as assume that we have a sample of events $\mathbf{x}_i$ with true categories t_i.
All events have the same initial weight $w_i^{(1)} = 1/N$

The iterative procedure looks like follows. In step j:

1. train classifier C_j using our input data $\mathbf{x}_i$ with weights $w_i^{(j)}$
2. calculate classifier response: $y_i^{(j)} = C_j(\mathbf{x}_i)$
3. calculate classifier error rate: $\varepsilon_j = \sum w_i^{(j)} \cdot (y_i^{(j)} != t_i) \; / \sum w_i^{(j)}$
4. calculate classifier weight: $a_j = \log\left(\frac{1-\varepsilon_j}{\varepsilon_j}\right)$

Procedure

Let as assume that we have a sample of events $\mathbf{x}_i$ with true categories t_i.
All events have the same initial weight $w_i^{(1)} = 1/N$

The iterative procedure looks like follows. In step j:

1. train classifier C_j using our input data $\mathbf{x}_i$ with weights $w_i^{(j)}$
2. calculate classifier response: $y_i^{(j)} = C_j(\mathbf{x}_i)$
3. calculate classifier error rate: $\varepsilon_j = \sum w_i^{(j)} \cdot (y_i^{(j)} != t_i) \; / \sum w_i^{(j)}$
4. calculate classifier weight: $a_j = \log\left(\frac{1-\varepsilon_j}{\varepsilon_j}\right)$
5. modify event weights:

$$w_i^{(j+1)} = w_i^{(j)} \cdot \exp(a_j) \quad \text{for } y_i^{(j)} \neq t_i,$$
$$w_i^{(j+1)} = w_i^{(j)} \quad \text{for } y_i^{(j)} = t_i.$$

Scale all weights to get $\sum w_i^{(j+1)} = 1$

Procedure

Let as assume that we have a sample of events $\mathbf{x}_i$ with true categories t_i.
All events have the same initial weight $w_i^{(1)} = 1/N$

The iterative procedure looks like follows. In step j:

1. train classifier C_j using our input data $\mathbf{x}_i$ with weights $w_i^{(j)}$
2. calculate classifier response: $y_i^{(j)} = C_j(\mathbf{x}_i)$
3. calculate classifier error rate: $\varepsilon_j = \sum w_i^{(j)} \cdot (y_i^{(j)} != t_i) \; / \sum w_i^{(j)}$
4. calculate classifier weight: $a_j = \log\left(\frac{1-\varepsilon_j}{\varepsilon_j}\right)$
5. modify event weights:

   or $\quad w_i^{(j+1)} = w_i^{(j)} \cdot \exp(-\alpha \; y_i^{(j)} \; t_i \; a_j)$.

Scale all weights to get $\sum w_i^{(j+1)} = 1$

Procedure (Behnke)

By reweighting events, we force subsequent classifiers to focus on events (i.e. value ranges) where classification was poor.

New classifiers are still "weak", but they properly classify different classes of events.

We get a sequence of classifiers focusing on different variable regions.

Procedure (Behnke)

By reweighting events, we force subsequent classifiers to focus on events (i.e. value ranges) where classification was poor.

New classifiers are still "weak", but they properly classify different classes of events.

We get a sequence of classifiers focusing on different variable regions.

We can get much stronger classifier by combining their outputs

$$C_{Boost}(\mathbf{x}) \;=\; \frac{1}{M}\sum_j a_j \; C_j(\mathbf{x})$$

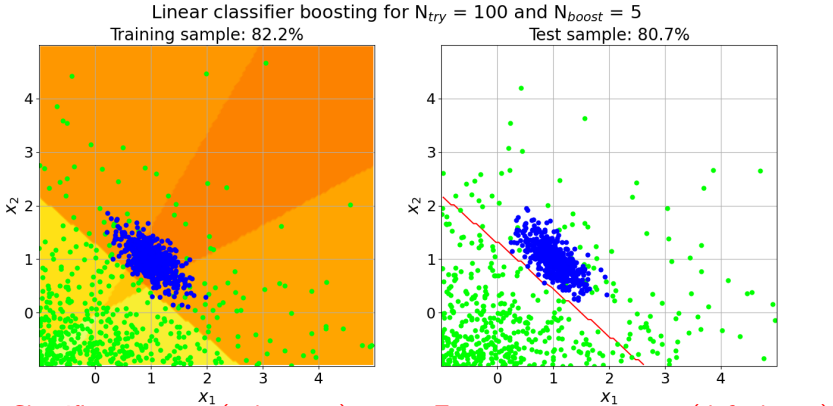
where M is the total number of classifiers in the collection.

This procedure is referred to as "adaptive boost" (AdaBoost)

Classifier boosting

13_Boost.ipynb Open in Colab

Example of weak classifier (linear discriminant) boosting



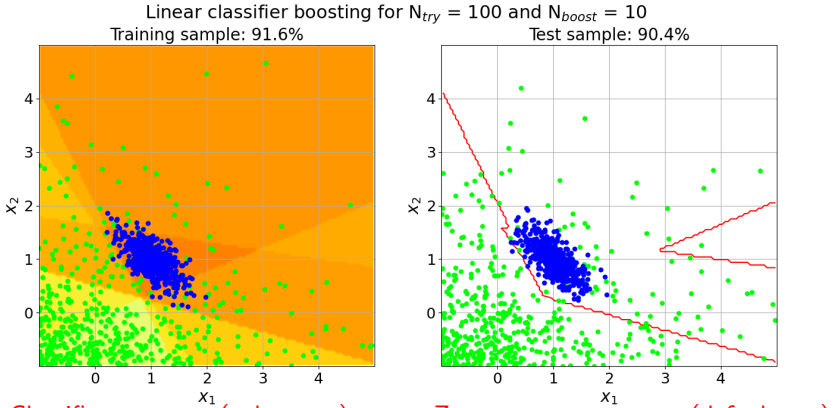
Classifier response (color map)

Zero response contour (default cut)

Classifier boosting

13_Boost.ipynb Open in Colab

Example of weak classifier (linear discriminant) boosting



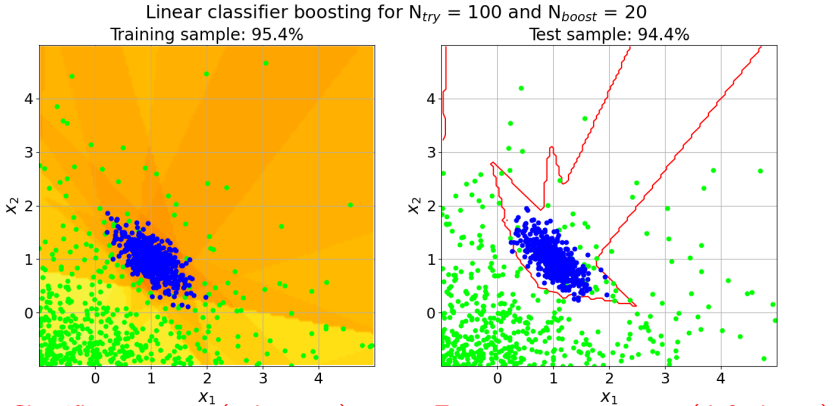
Classifier response (color map) Zero response contour (default cut)

Classifier boosting

13_Boost.ipynb

Open in Colab

Example of weak classifier (linear discriminant) boosting



Classifier response (color map)

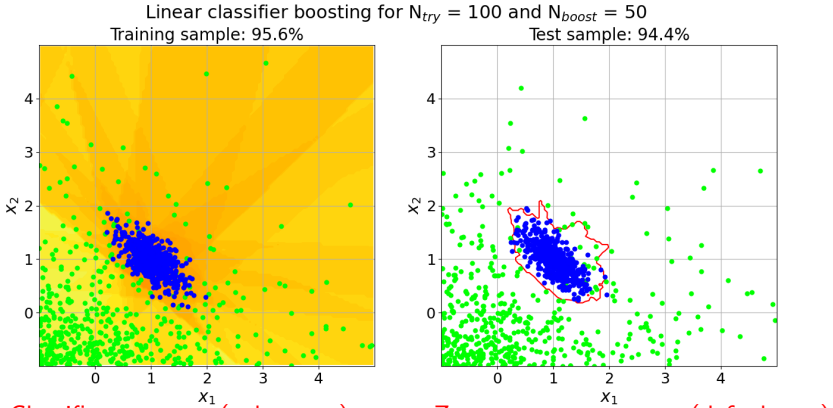
Zero response contour (default cut)

Classifier boosting

13_Boost.ipynb

Open in Colab

Example of weak classifier (linear discriminant) boosting



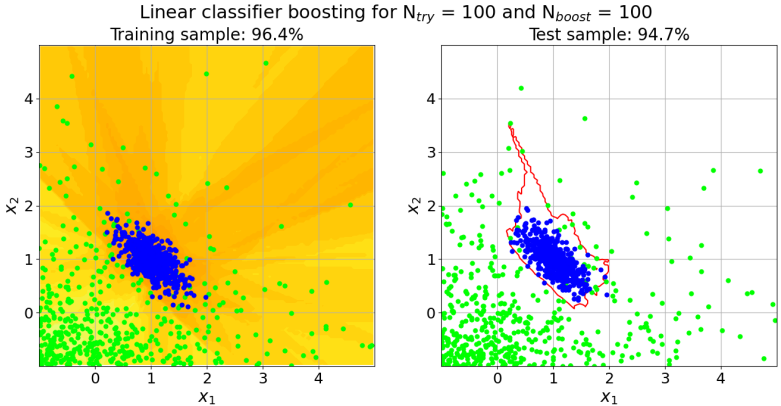
Classifier response (color map)

Zero response contour (default cut)

Classifier boosting

13_Boost.ipynb Open in Colab

Example of weak classifier (linear discriminant) boosting

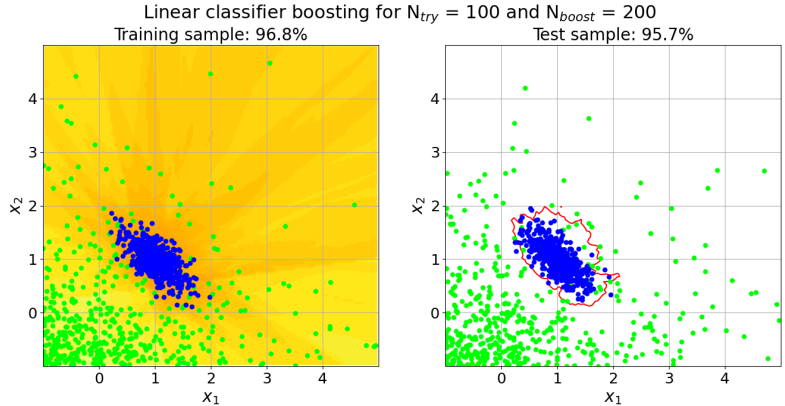


Final performance improves significantly $\Rightarrow$ "strong classifier" obtained

Classifier boosting

13_Boost.ipynb Open in Colab

Example of weak classifier (linear discriminant) boosting

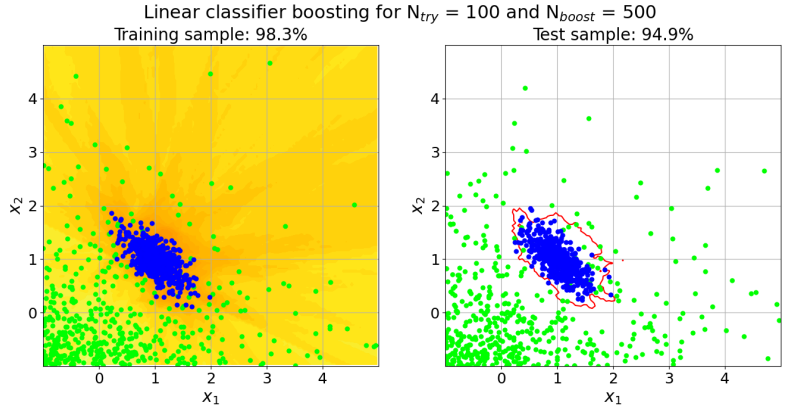


Final performance improves significantly $\Rightarrow$ "strong classifier" obtained

Classifier boosting

13_Boost.ipynb Open in Colab

Example of weak classifier (linear discriminant) boosting

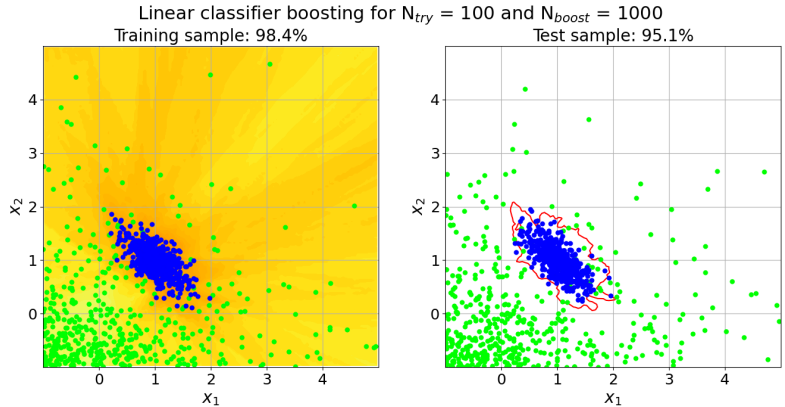


Final performance improves significantly $\Rightarrow$ "strong classifier" obtained

Classifier boosting

13_Boost.ipynb Open in Colab

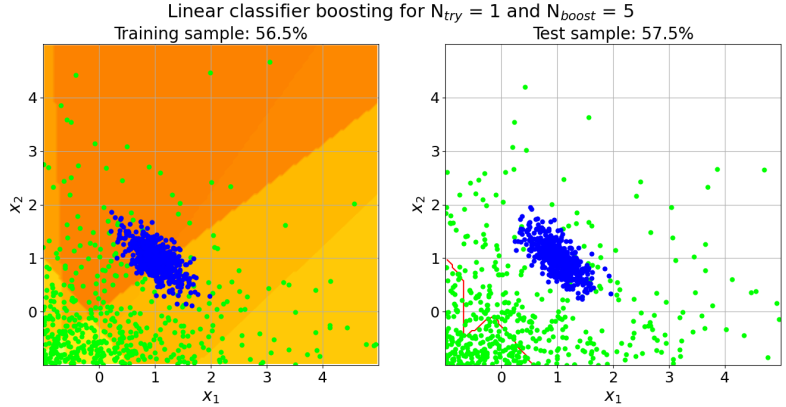
Example of weak classifier (linear discriminant) boosting



Final performance improves significantly $\Rightarrow$ "strong classifier" obtained

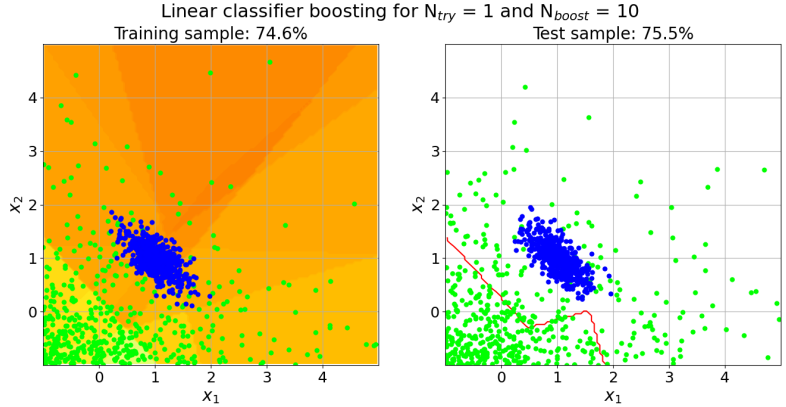
Classifier boosting

Surprisingly, the procedure works also for the completely random (not optimized in any way) classifiers used as "building blocks"



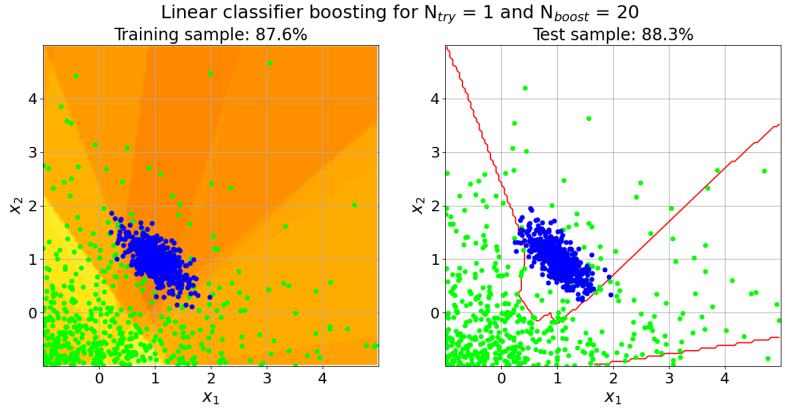
Classifier boosting

Surprisingly, the procedure works also for the completely random (not optimized in any way) classifiers used as "building blocks"



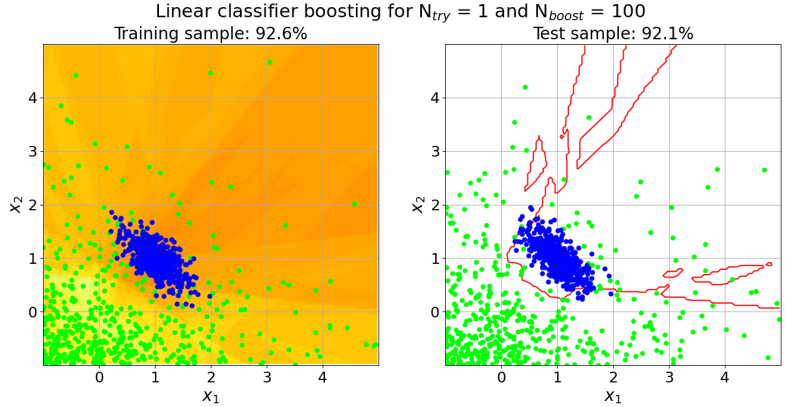
Classifier boosting

Surprisingly, the procedure works also for the completely random (not optimized in any way) classifiers used as "building blocks"



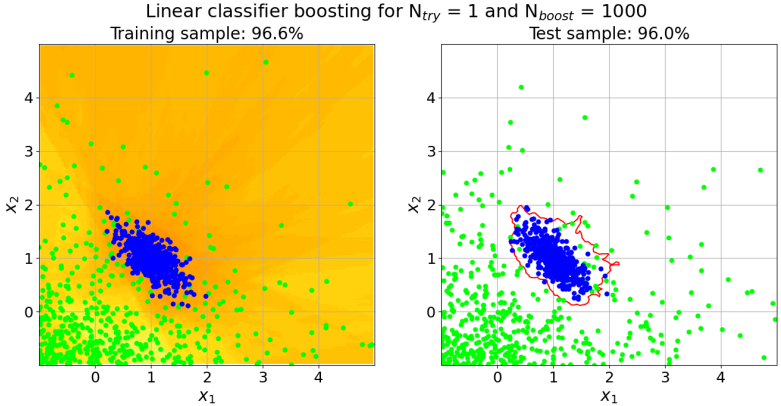
Classifier boosting

Surprisingly, the procedure works also for the completely random (not optimized in any way) classifiers used as "building blocks"



Classifier boosting

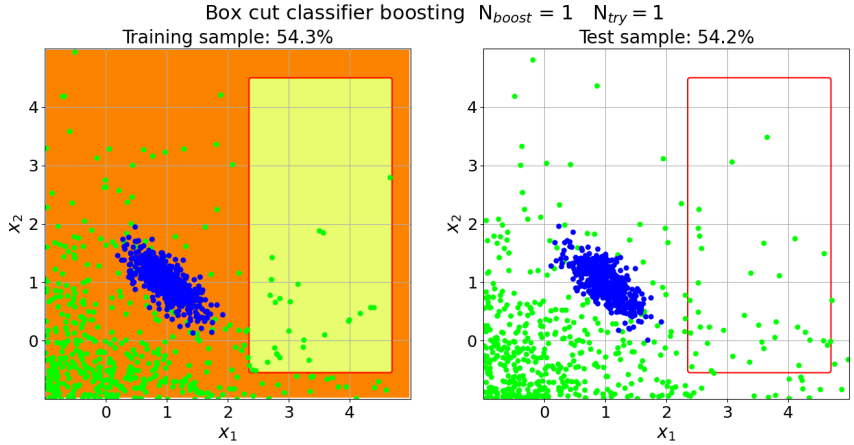
Surprisingly, the procedure works also for the completely random (not optimized in any way) classifiers used as "building blocks"



Final performance only slightly worse than for more optimized input...

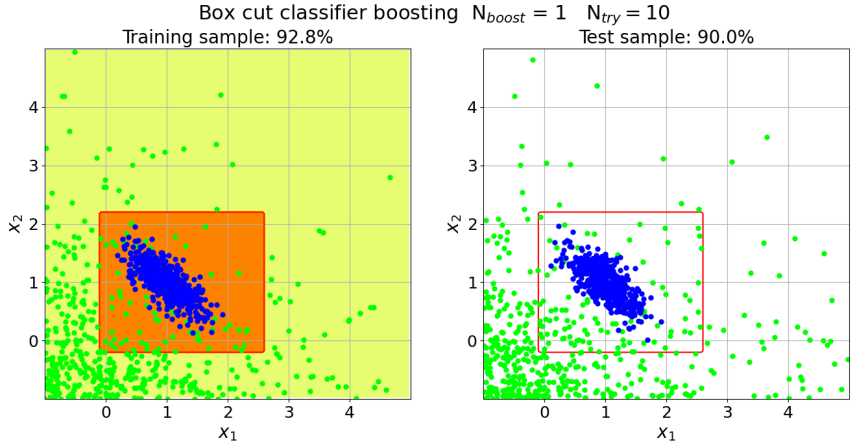
Box cut classifier

Random box cut based on two random points in the parameter space:



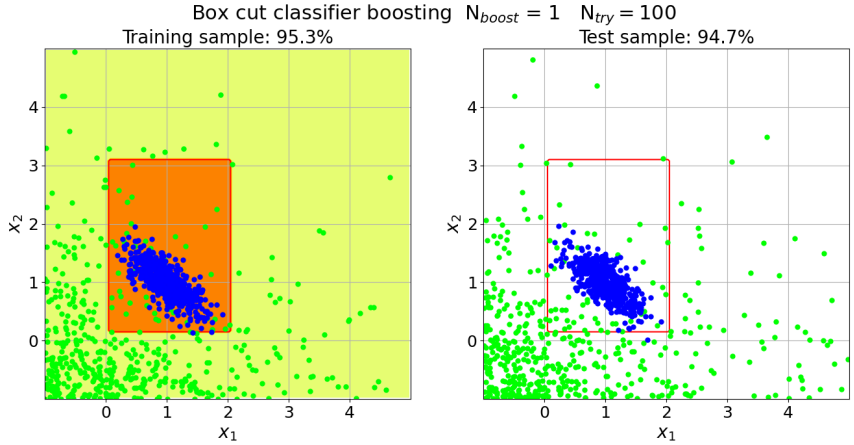
Box cut classifier

Box cut with highest efficiency selected out of 10 random box cuts



Box cut classifier

Box cut with highest efficiency selected out of 100 random box cuts

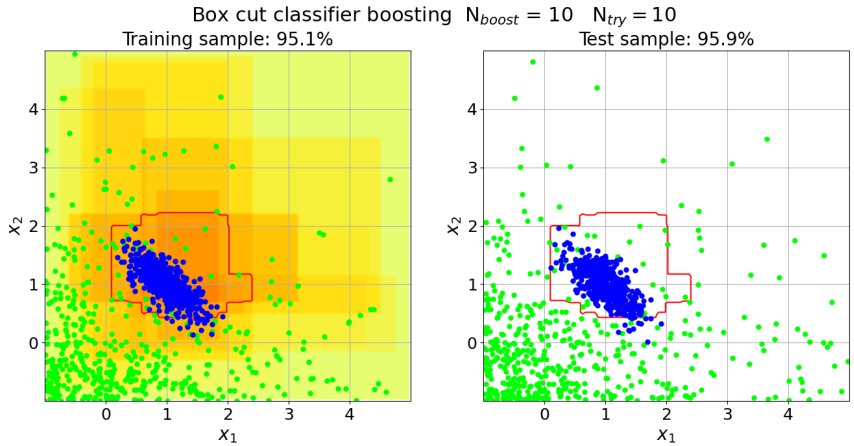


Box classifier boosting

13_Cuts.ipynb

Open in Colab

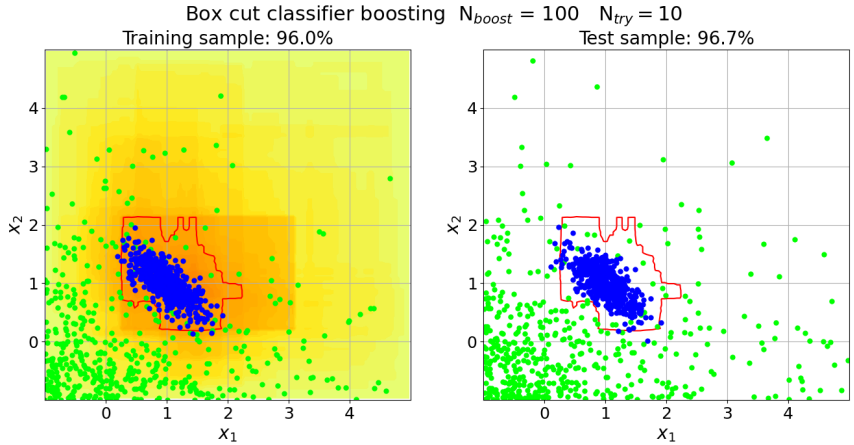
Example of weak classifier (best box cut out of 10 random) boosting



Box classifier boosting

13_Cuts.ipynb Open in Colab

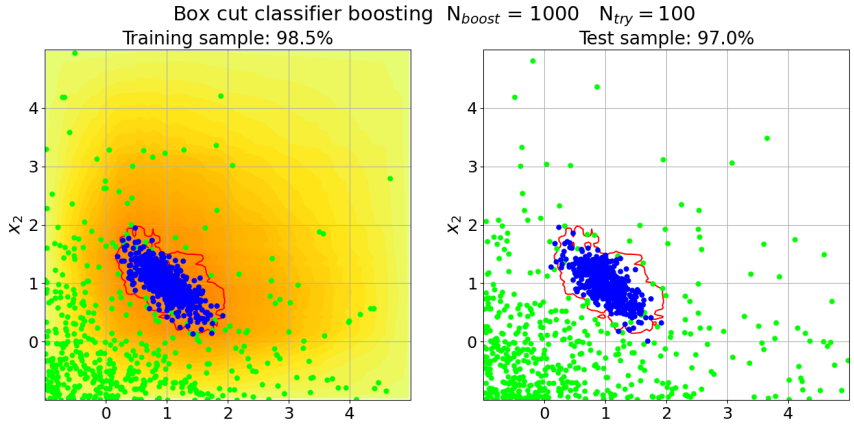
Example of weak classifier (best box cut out of 10 random) boosting



Box classifier boosting

13_Cuts.ipynb Open in Colab

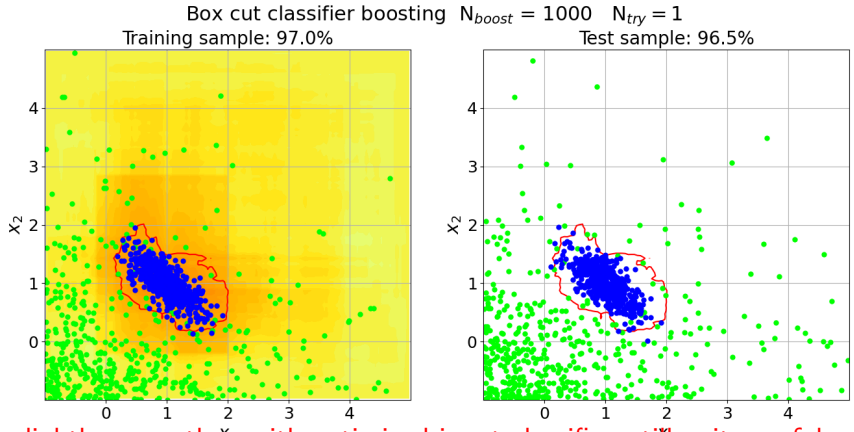
Example of weak classifier (best box cut out of 100 random) boosting



Selection closely follows train data! Worse for test data, but still very efficient...

Box classifier boosting

Even random box cut (without selection) can get boosted



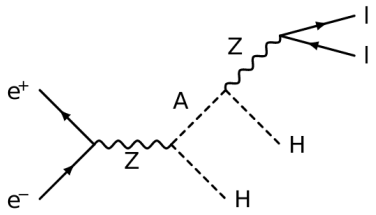
Results only slightly worse than with optimized input classifier, still quite useful...

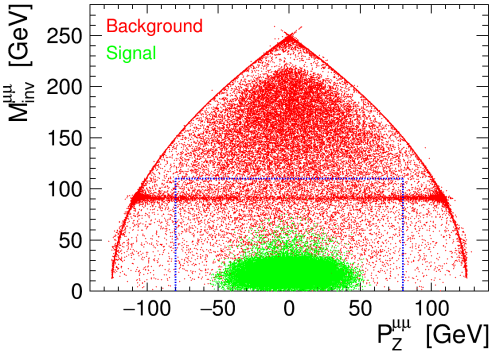
Machine Learning

Principle

It is quite a common approach in data selection to apply cuts on variables considered.
We can profit from our understanding of the processes studied...

IDM scalar pair-production
with di-lepton signature

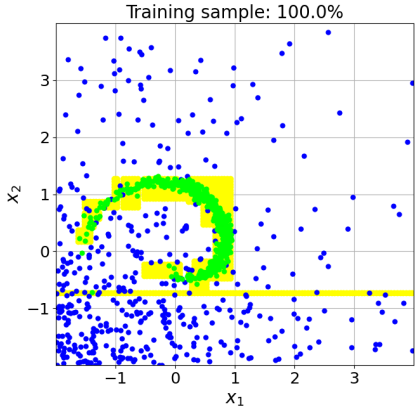




However, tuning the cuts by hand is difficult...

Example

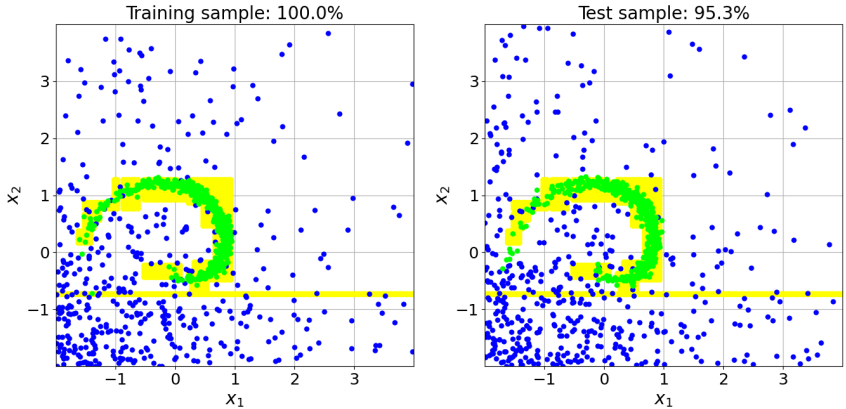
We can write down the cuts that will perfectly classify our training sample:



Example

But on test sample results will be worse!

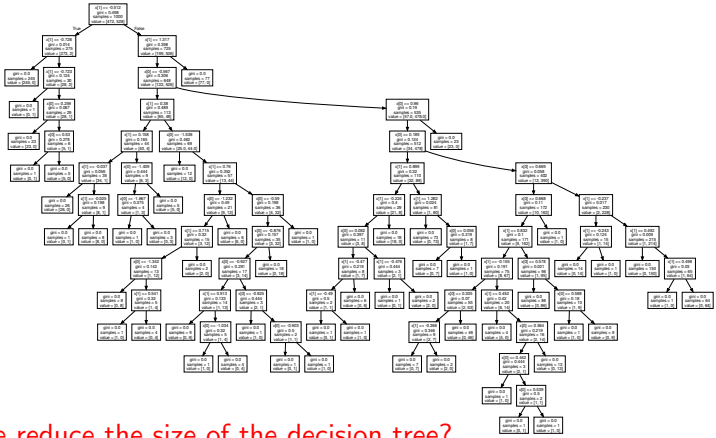
Efficiency ∼95%



Note that this will get much poorer in multi-dimensional space...

Example

The tree for full sample classification very complicated already in 2-D...



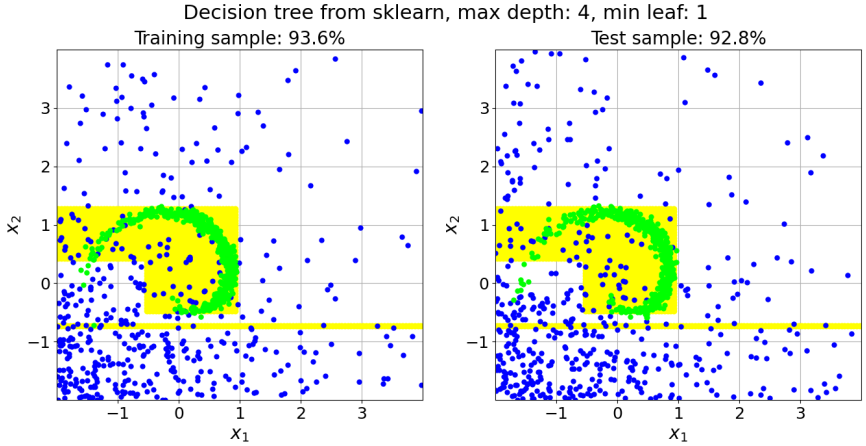
How much can we reduce the size of the decision tree?

Example

13_skTree.ipynb

Open in Colab

Good performance (efficiency above 90%) already for 4 cut levels!

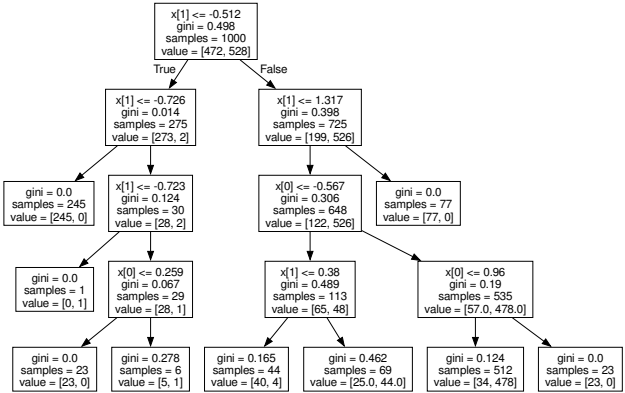


Example

13_skTree.ipynb

Open in Colab

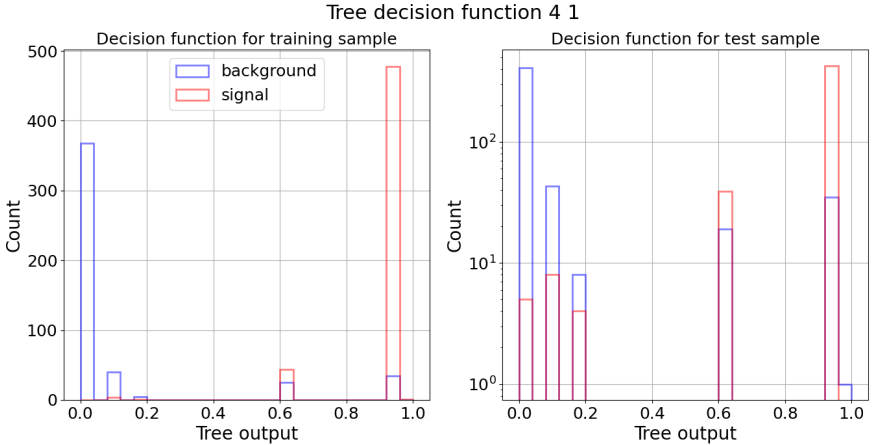
Good performance (efficiency above 90%) already for 4 cut levels!



Example

13_skTree.ipynb Open in Colab

Good performance (efficiency above 90%) already for 4 cut levels!



Machine Learning

Boosted Decision Trees

For their good performance, decision trees are "natural candidates" for use in boosting procedure, to get even better classifiers.

Boosted Decision Trees (BDT) algorithms are widely used in particle physics, mainly for their flexibility and stability.

Boosted Decision Trees

For their good performance, decision trees are "natural candidates" for use in boosting procedure, to get even better classifiers.

Boosted Decision Trees (BDT) algorithms are widely used in particle physics, mainly for their flexibility and stability.

Many different algorithms exist, both concerning tree generation and training, and boosting procedure.

Wide range of options implemented in **sklearn** library.

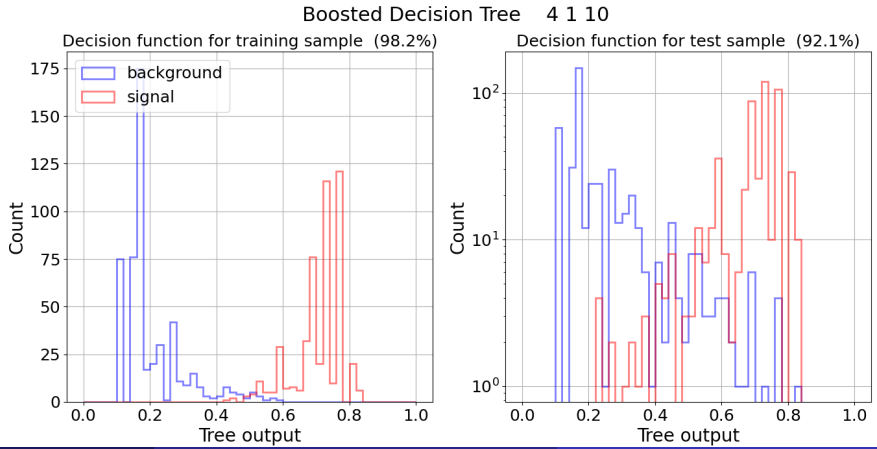
TMVA (Multi Variate Analysis) package for **root** widely used in particle physics community. More advanced tuning options ($\Rightarrow$ better performance?), but more complicated to use. Based on root, is well integrated into data processing and analysis framework...

BDT Example

13_skBDT.ipynb Open in Colab

Good performance (efficiency $> 90\%$) already with 10 trees.

10 trees

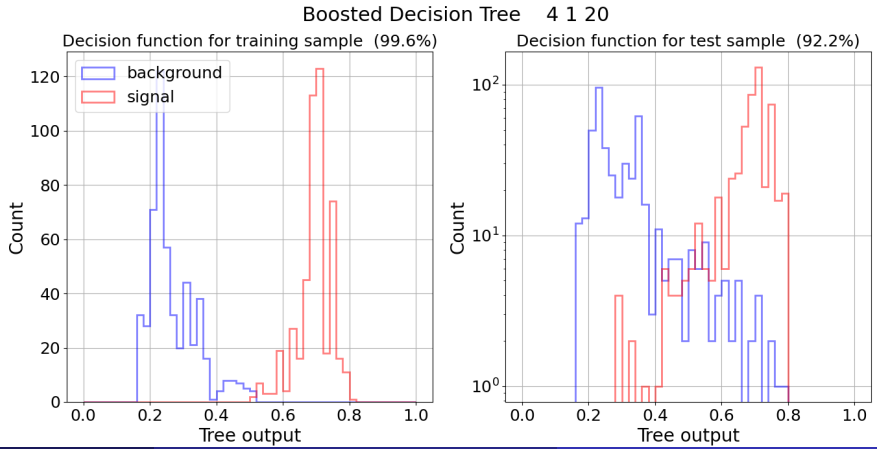


BDT Example

13_skBDT.ipynb Open in Colab

Good performance (efficiency $> 90\%$) already with 10 trees.

20 trees

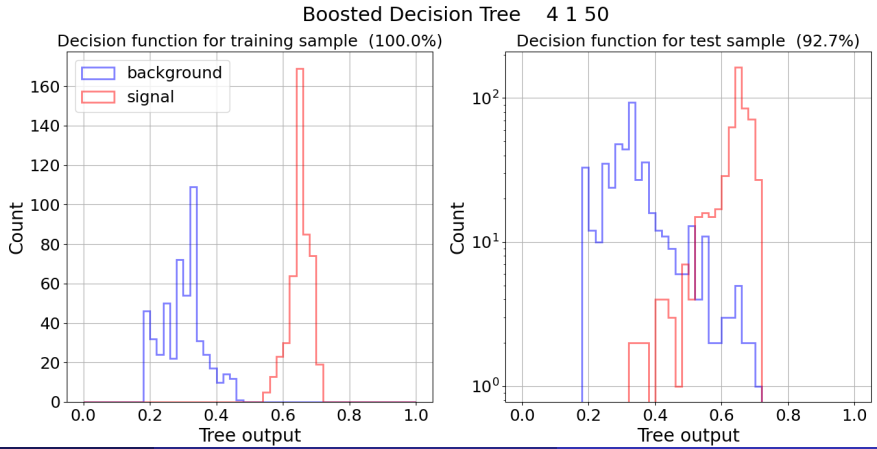


BDT Example

13_skBDT.ipynb Open in Colab

Good performance (efficiency $> 90\%$) already with 10 trees.

50 trees

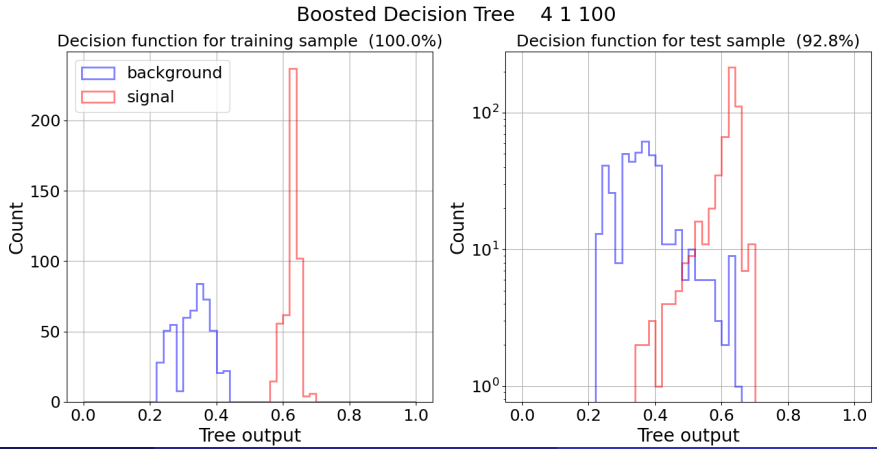


BDT Example

13_skBDT.ipynb Open in Colab

Good performance (efficiency $> 90\%$) already with 10 trees.

100 trees

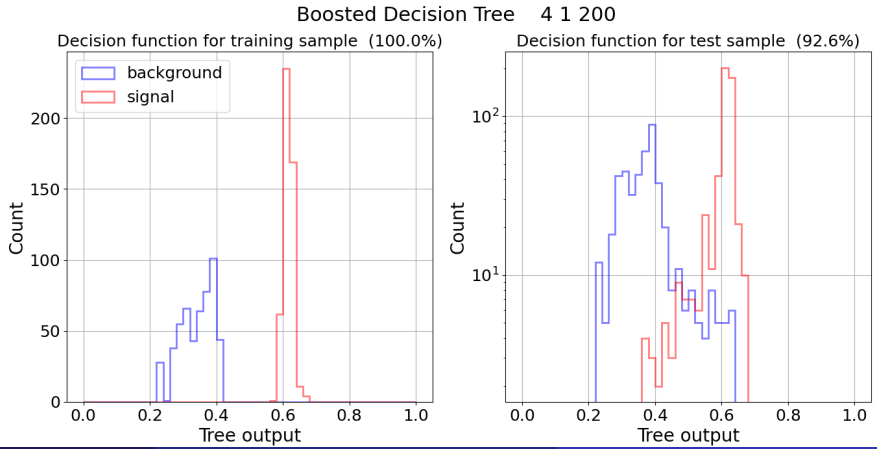


BDT Example

13_skBDT.ipynb Open in Colab

Good performance (efficiency $> 90\%$) already with 10 trees.

200 trees

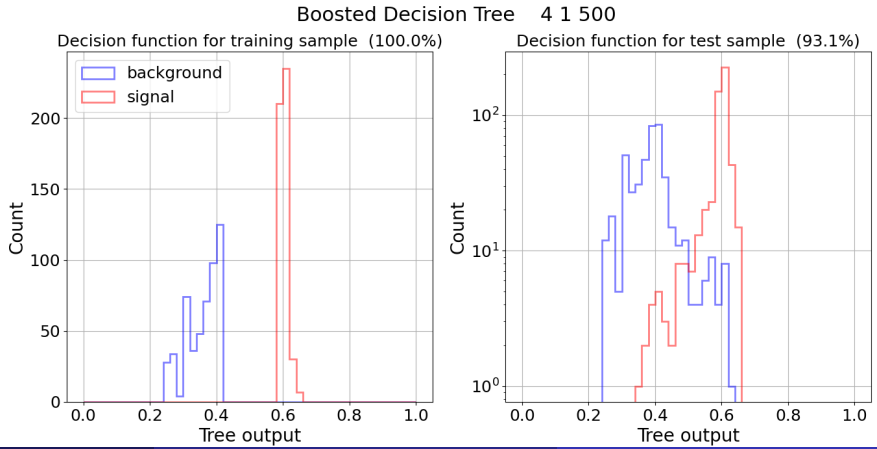


BDT Example

13_skBDT.ipynb Open in Colab

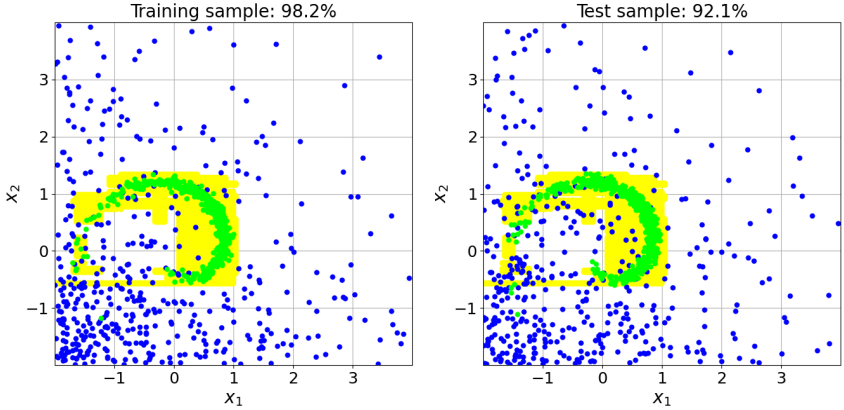
Good performance (efficiency $> 90\%$) already with 10 trees.

500 trees



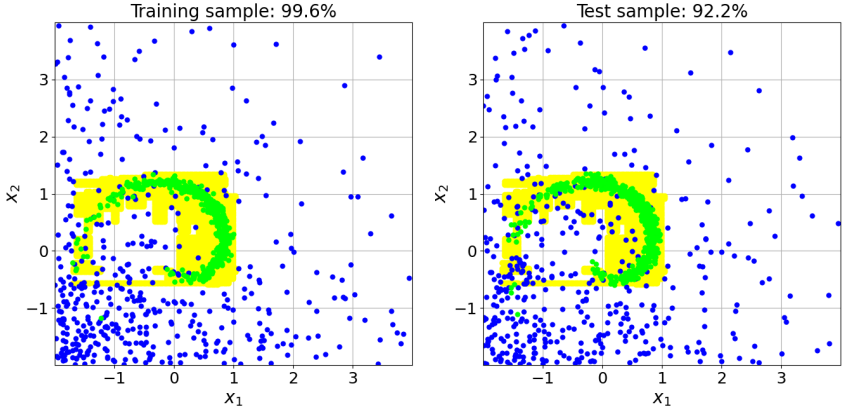
BDT Example

Classification "follows" training sample better and better, with increasing number of trees...
But results "saturate" at some point (at efficiency $\sim 93\%$) for independent test sample.
10 trees



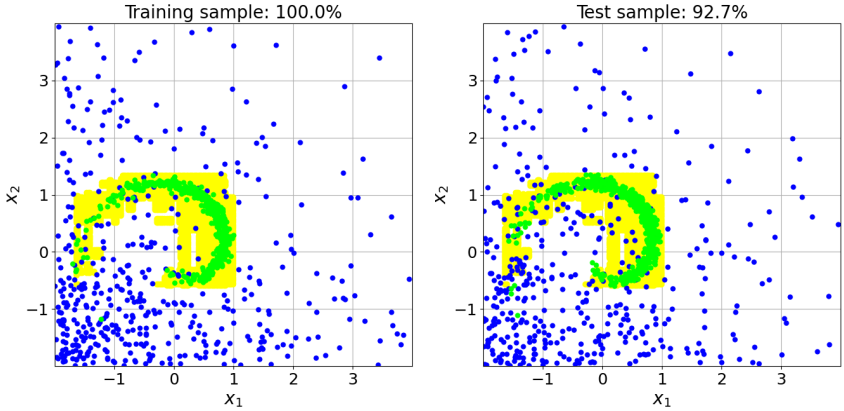
BDT Example

Classification "follows" training sample better and better, with increasing number of trees...
But results "saturate" at some point (at efficiency $\sim 93\%$) for independent test sample.
20 trees



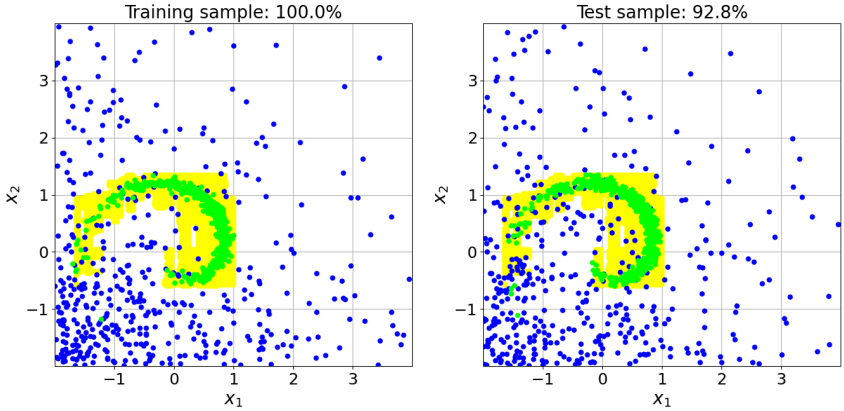
BDT Example

Classification "follows" training sample better and better, with increasing number of trees...
But results "saturate" at some point (at efficiency $\sim 93\%$) for independent test sample.
50 trees



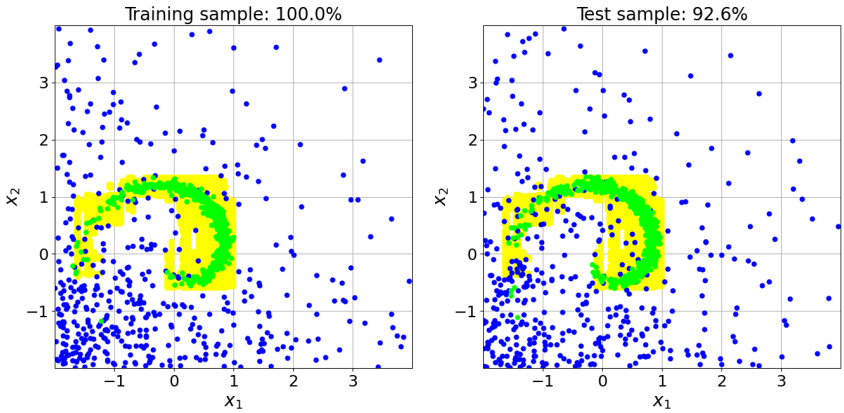
BDT Example

Classification "follows" training sample better and better, with increasing number of trees...
But results "saturate" at some point (at efficiency $\sim 93\%$) for independent test sample.
100 trees



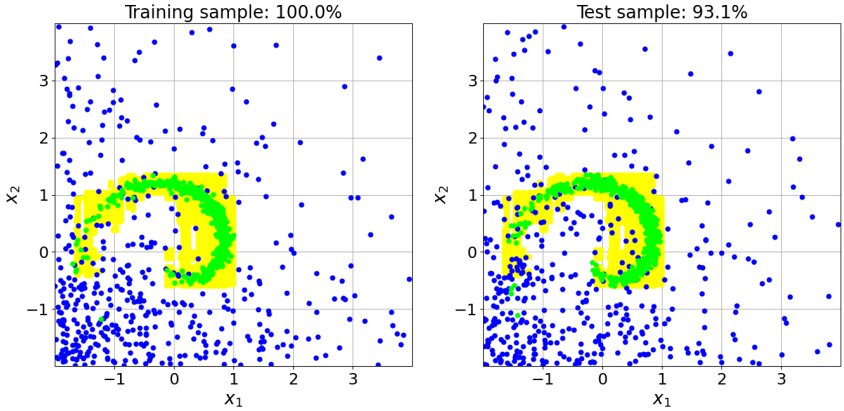
BDT Example

Classification "follows" training sample better and better, with increasing number of trees...
But results "saturate" at some point (at efficiency $\sim 93\%$) for independent test sample.
200 trees



BDT Example

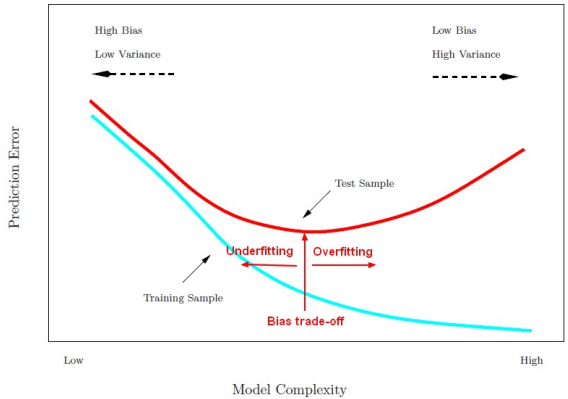
Classification "follows" training sample better and better, with increasing number of trees...
But results "saturate" at some point (at efficiency $\sim 93\%$) for independent test sample.
500 trees



Overtraining

source: datacadamia.com

Is a common problem in all Machine Learning methods

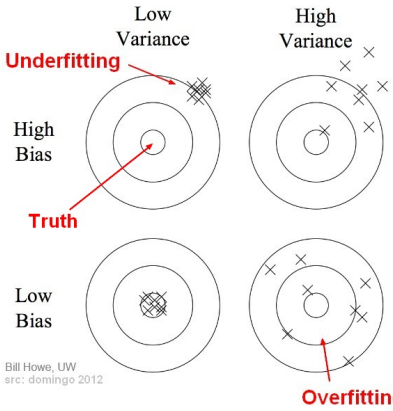


If we try too hard (also by using too many variables !), result can get worse...

Overtraining

source: datacadamia.com

Is a common problem in all Machine Learning methods



If we try too hard (also by using too many variables !), result can get worse...

Machine Learning

Homework

Solutions to be uploaded by January 29.

Three samples of events $\mathbf{x} = (x_1, x_2, x_3, x_4)$ were prepared:

- training signal sample
- training background sample
- test sample, with both signal and background events, for the actual analysis

$\Rightarrow$ to be downloaded from the lecture web page 13_homework_read.ipynb Open in Colab

Use one of the presented approaches to obtain event classification for the considered event samples:

- draw ROC curve for the obtained classifier
- extract the fraction of the signal events in the test sample (with uncertainty)
- check how the result and its uncertainty depend on the classifier response cut

Numbers of signal and background events selected from the test sample have to be corrected for classification efficiency and errors...

Homework data

13_homework_read.ipynb

Open in Colab