Machine Learning for Classification.

A Modern Theory of Detection and Estimation.
Block-2: Detection

Emilio Parrado-Hernández, emilio.parrado@uc3m.es

November 7, 2022

Index

Index

Detection problems defined in terms of data

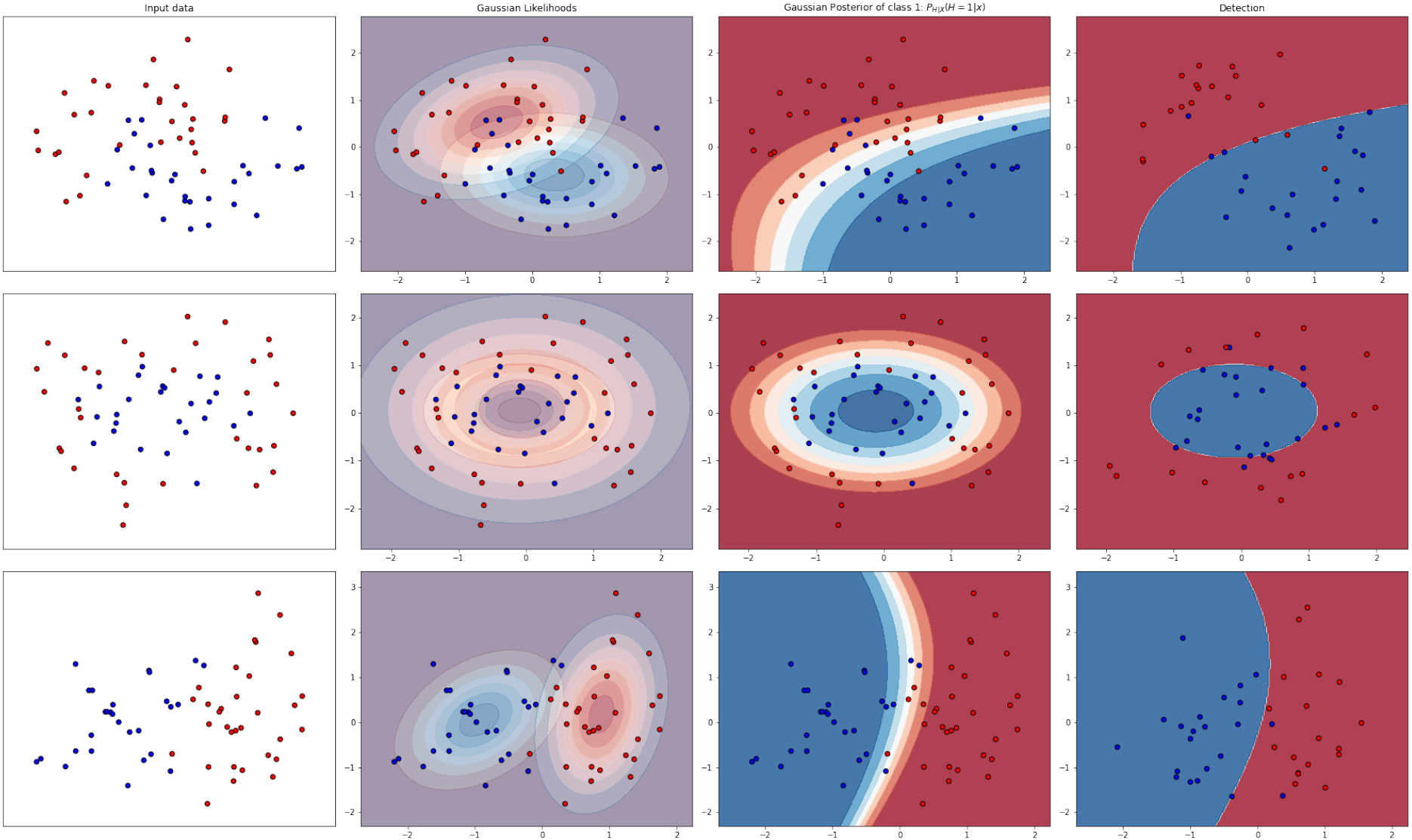
- **Detection**
  - Several **hypotheses** generate observations $\Rightarrow$ each observation comes from one of these hypothesis
  - **Detector** finds out which hypothesis was responsible for each observation
  - Statistical characterization:
    - Prior $P(H = h)$, $h = 1, \ldots, L$
    - Likelihood: $p_{X|H}(\mathbf{x}|h)$, $h = 1, \ldots, L$
    - **costs** c_{dh}, $d, h = 1, \ldots, L$
    - **observations are random** until we get a test observation to form the **posterior** $P_{H|X}(h|\mathbf{x})$, $h = 1, \ldots, L$
- **Classification**
  - **Data collection** of N pairs (observation, target) $\{(\mathbf{x}_i, y_i)\}_{i=1}^{N}$
  - Classes can be assimilated to hypotheses generating the data; $y_i \in \{1, \ldots, L\}$ is the true class of observation $\mathbf{x}_i$
  - These N data pairs are not random, we **have observed** them
  - **Classifier** finds out the **true class** of every test observation (those test observations are random)

Analytic, semi-analytic and machine methods

Three strategies to solve detection problems. Solving a detection problem means **find a discriminant function** to classify (find true hypothesis for) test observations

- The problem is defined in terms of **Statistics** (prior and likelihoods): Use **Detection Theory** to find the discriminat function
- The problem is defined with a dataset:
  - **semi-analytic methods**: Use the data to **estimate** the priors and the likelihoods and then apply **detection theory** with these *learned* pdfs to get the discriminant function
  - **Machine Learning**: Use the data to **fit** a discriminant function able to produce correct classifications, but without aiming at learning the true joint pdf.

Semi-analytic methods example



Machine Learning

So far we have studied how to construct detectors using a mathematical model of the problem we need to solve: likelihoods, priors, posteriors, costs, etc
The common situation in real life is that problems are defined by sets of examples. In fact, human beings are most of the time learning by examples.

Machine learning pipeline

1. Receive training set: observations and targets (true class of each observation)
2. Choose a family of models
3. Choose hyperparameters: control the optimization of the model
4. Fit the model with the training data (optimization)
5. Receive the test set: observations without targets
6. Predict a target for each test observation with the model
7. Receive the test set true targets and **evaluate** the accuracy of the model

Training set and test set

The **training set** is formed by N pairs $\{(\mathbf{x}_i, y_i)\}_{i=1}^{N}$. The **test set** is formed by N_t pairs $\{(\mathbf{x}_t, y_t)\}_{t=1}^{N_t}$.

The training and test observations are assumed to be drawn iid from a same data probability distribution.

- Observations $\mathbf{x}_i$ live in a d-dimensional space $\mathbf{x}_i \in \mathbb{R}^d$
- targets are discrete and finite. They take the role of hypotheses in analytic detection. In multiclass problems we usually use integers to denote the different classes. In binary classification we usually employ $y_i \in \{-1, 1\}$ (negative and positive classes) or $y_i \in \{0, 1\}$

Data Matrices

$$X = \begin{bmatrix} x_{1,1} & x_{1,2} & \cdots & x_{1,d} \\ x_{2,1} & x_{2,2} & \cdots & x_{2,d} \\ \vdots & \vdots & \ddots & \vdots \\ x_{N,1} & x_{N,2} & \cdots & x_{N,d} \end{bmatrix} \quad \mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_N \end{bmatrix}$$

Each row of X stores an observation

Generalization and overfitting

The goal of machine learning is to learn a model capable of achieving a great **classification accuracy in the test set**.

In general learning to classify correctly the training set is a good strategy to correctly classify those test observations not used for fitting the model

However, the optimization that fits the model can incur in **overfitting**. This phenomenon happens when the performance achieved in the training set is much better than the performance experimented in the test set.

This generalization with the test set is related to the concept of expected risk of a classifier: The expected error rate in a test observation is the **true risk** of a machine learning classifier. In machine learning the average error rate in the training set is called **empirical risk**. Notice the empirical risk can always be measured, but in most of the cases the true risk can only be estimated.

Along the course we will present several ways to estimate the true risk, like using a **separate test dataset** or **crossvalidation**

Index

Parametric and non-parametric methods

The core of the machine learning pipeline is to fit the parameters of the model. This is equivalent to coming up with the **discriminant function** $g(\mathbf{x})$.

Typically a model is a software template with **free parameters**. **Learning** involves an optimization that uses the training data to set these free parameters to values that endow the model with the best possible generalization capability.
Models can be classified into two great groups according to the number of parameters to fit:

- **Parametric models**: The number of parameters is fixed before obtaining the training data. For instance a linear classifier with $4-$dimensional data needs to fix 5 parameters independently of the size of the training set
- **Non-parametric models**: The number of parameters is a function of the training dataset size. Some examples at the end of the notes

Index

Index

kNN outline

- Non parametric model. There is one **hyperparameter**, k, number of neighbors.
- Underlying assumption: If two observations $\mathbf{x}_i$ and $\mathbf{x}_j$ are close enough, it is reasonable to think they will belong to the same class.
- Locality classification: Each test observation is classified exclusively according to its neighbors: closest observations in the training set.
- The value of k determines the size of the neighborhood, and controls the locality of the classifier:
  - $k = 1$: each test observation is assigned to the class of its closest observation in the training set.
  - $k = N$: all the test observations are assigned to the **majority class**
  - intermediate values of k modulate the sizes of the neighborhood of each observation.
- The value of k has to be fixed with **prior domain knowledge**. The range of values of k that works well will be different for each problem. Notice the density of observations can vary across the input space. That is, for a test observation in a densely populated area of the input space its k nearest neighbors can fit in a ball of a smaller radius compared to the case of a test observation in a scarcely populated area.

kNN discriminant function

The discriminant function for kNN is constructed in an algorithmic way. For every observation $\mathbf{x}_t$ that you need to classify with kNN follow these steps:

1. Compute the **distance** between $\mathbf{x}_t$ and all the observations in the training set.
2. Sort these distances in ascending order
3. Find the training observations that occupy the first k positions in the ranking of distances to $\mathbf{x}_t$. Let's denote by $\mathcal{I}_k = \{i_1, i_2, \ldots, i_k\}$ the indices of these training observations in the training set.
4. Retrieve the true targets of the kNN: $\{y_{i_1}, y_{i_2}, \ldots, y_{i_k}\}$
5. $\hat{y}_t = \text{mode}\{y_{i_1}, y_{i_2}, \ldots, y_{i_k}\}$

The last step can be modified replacing the mode by a voting, where each neighbor $\mathbf{x}_{i_j}$ votes for its class with its vote inversely proportional to the distance between $\mathbf{x}_{i_j}$ and $\mathbf{x}_t$.

kNN model and posterior pdf

Uniform weighting of votes

Local approximation to the posterior pdf of each class

$$P_{H|X}(H = h|\mathbf{x}_t) \approx \frac{N_h}{k}$$

where N_h is the number of training observations of class h in the neighborhood of $\mathbf{x}_t$

Distance weighting of votes

Local approximation to the posterior pdf of each class

$$P_{H|X}(H = h|\mathbf{x}_t) = \frac{p_{X|H}(\mathbf{x}|H = h)P_H(h)}{p_X(\mathbf{x})}$$

$$p_{X|H}(\mathbf{x}|H = h)P_H(h) \approx \sum_{\mathbf{x}_i \in n_k(\mathbf{x}_t), y_i \in h} \frac{1}{d(\mathbf{x}, \mathbf{x}_i)}$$

where $d(\mathbf{x}, \mathbf{x}_i)$ is the distance between the test observation and each of the training observations in the neighborhood of $\mathbf{x}$

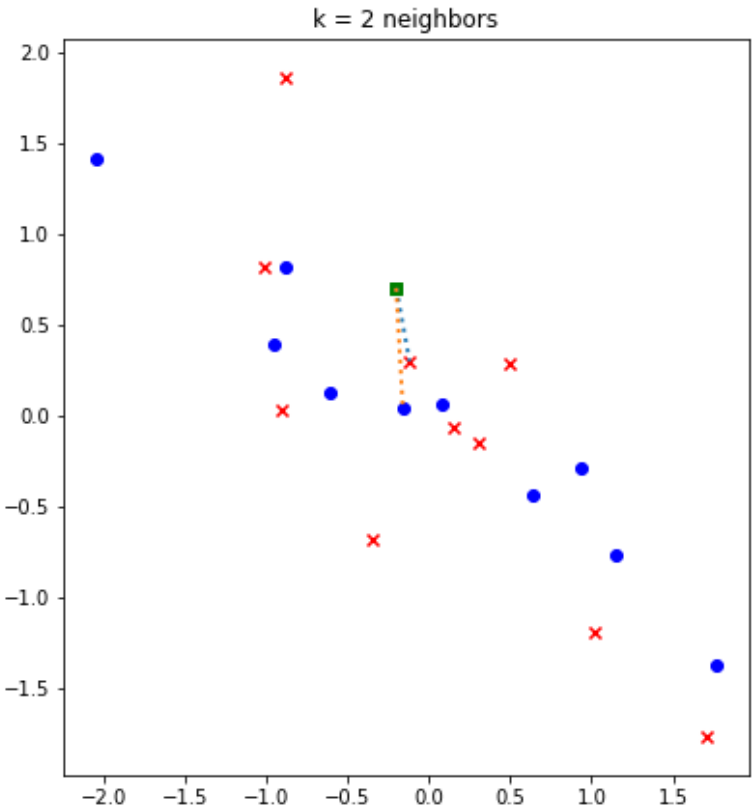
Learning the kNN model

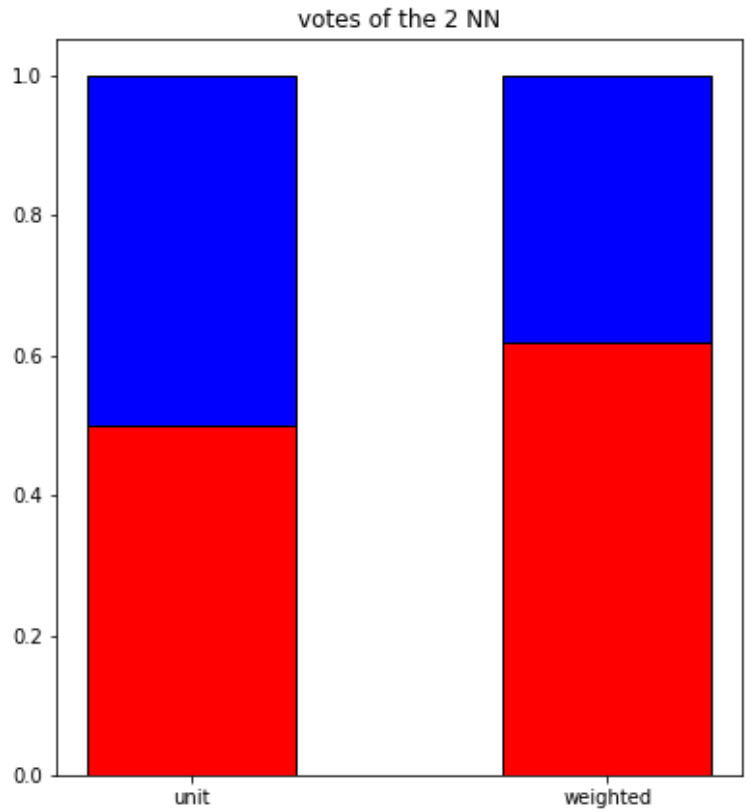
- In fact there is no such a learning process as with other machine learning models, where one need to find values for free parameters running an optimization with the training set observations. kNN just needs to setup an indexed **table that stores the training observations** and their **targets**.
- Choose a **distance** or **metric** to compare the observations, typically the **Euclidean distance** aka L_2 norm:

$$d(\mathbf{x}_1, \mathbf{x}_2) = \sqrt{\sum_{j=1}^{d} (x_{1,j} - x_{2,j})^2} = \sqrt{(\mathbf{x}_1 - \mathbf{x}_2)^\top (\mathbf{x}_1 - \mathbf{x}_2)}$$

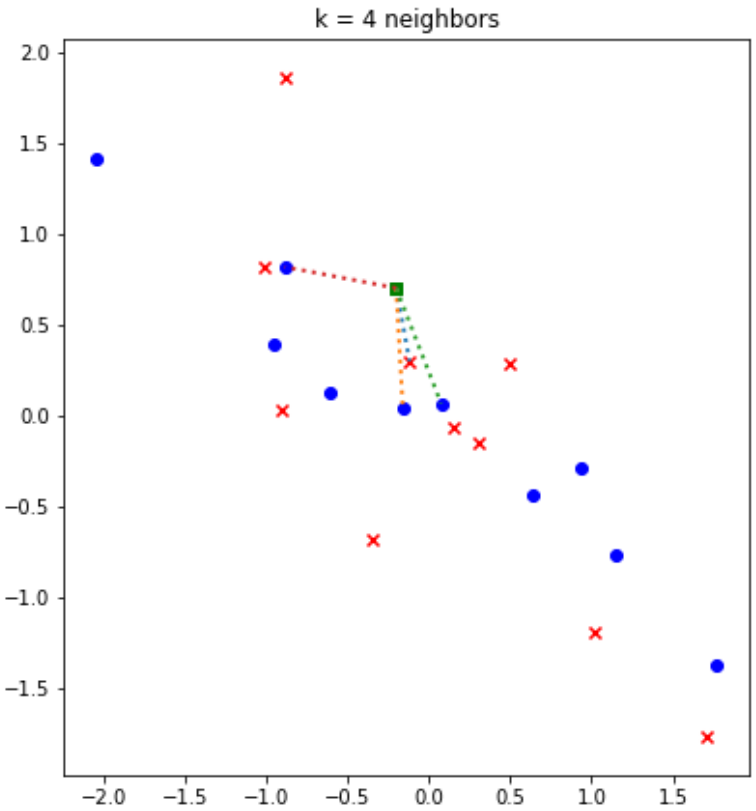
  in python `d_x1_x2 = numpy.linalg.norm(x1-x2)`
- Fix k (number of neighbors)
- Fix a voting strategy to determine the majority class from the k neighbors:
  - Each neighbor contributes a unit vote to its corresponding class (same as computing the mode of the targets)
  - Each neighbor vote is **weighted inversely proportional** to its distance to $\mathbf{x}_t$.

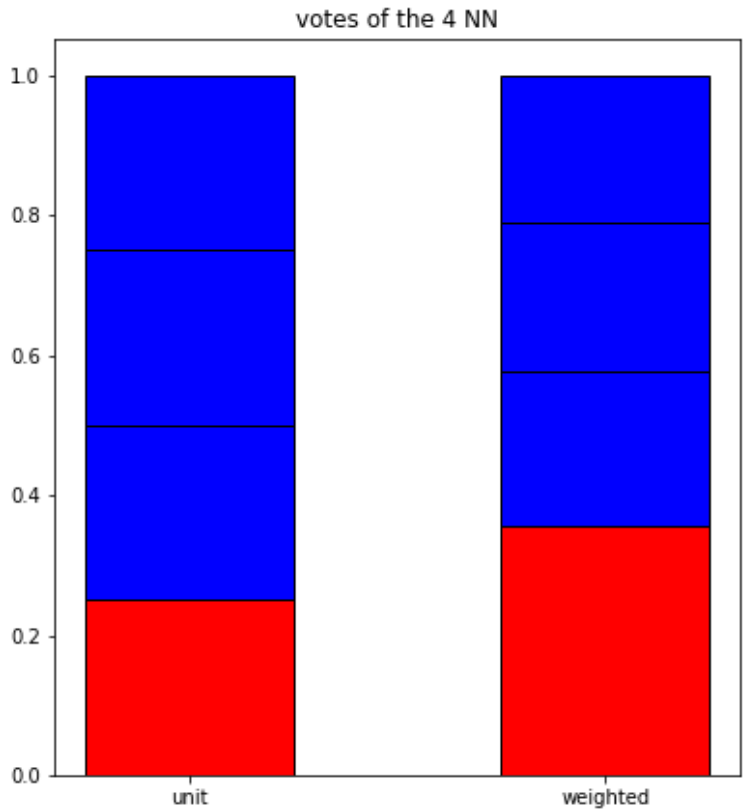
Example


k = 2 neighbors
2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5

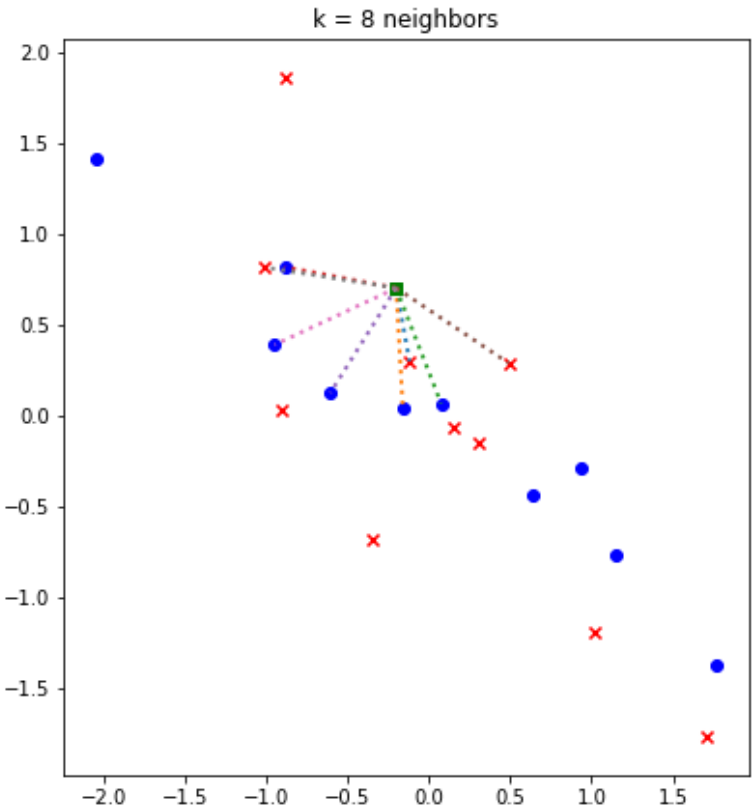

votes of the 2 NN
1.0
0.8
0.6
0.4
0.2
0.0
unit
weighted

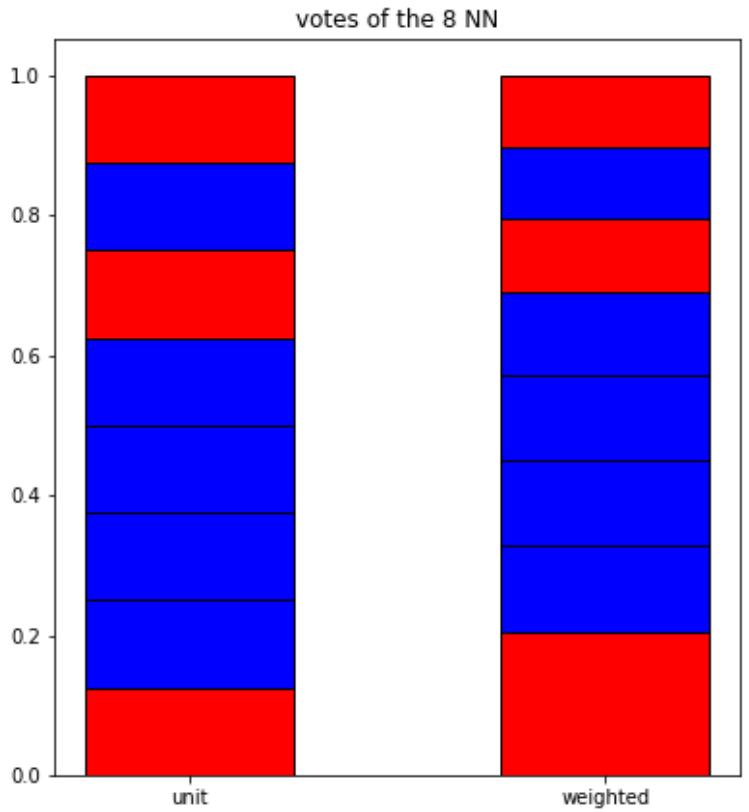
Example


k = 4 neighbors
2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5


votes of the 4 NN
1.0
0.8
0.6
0.4
0.2
0.0
unit
weighted

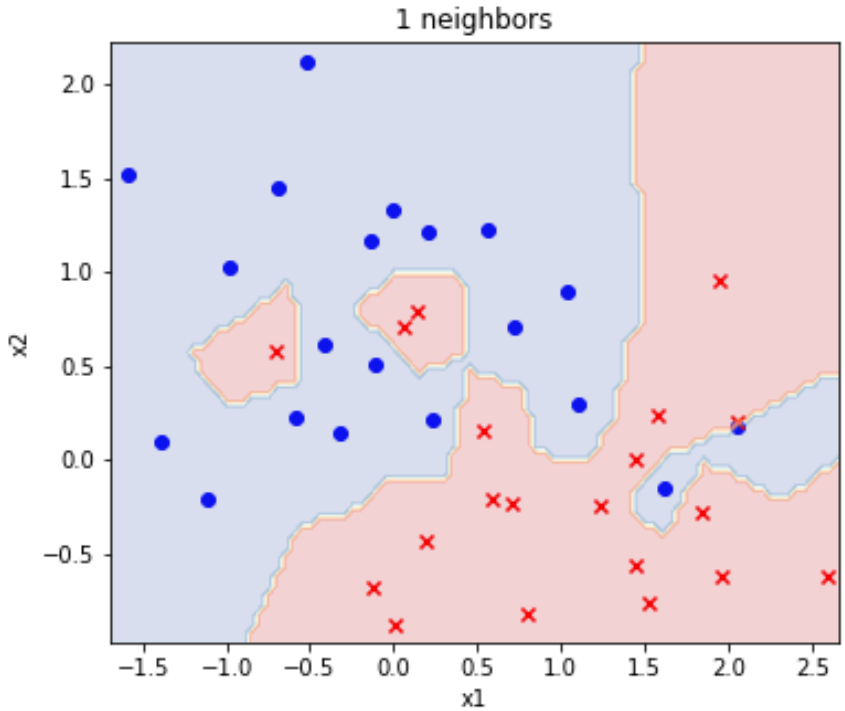
Example

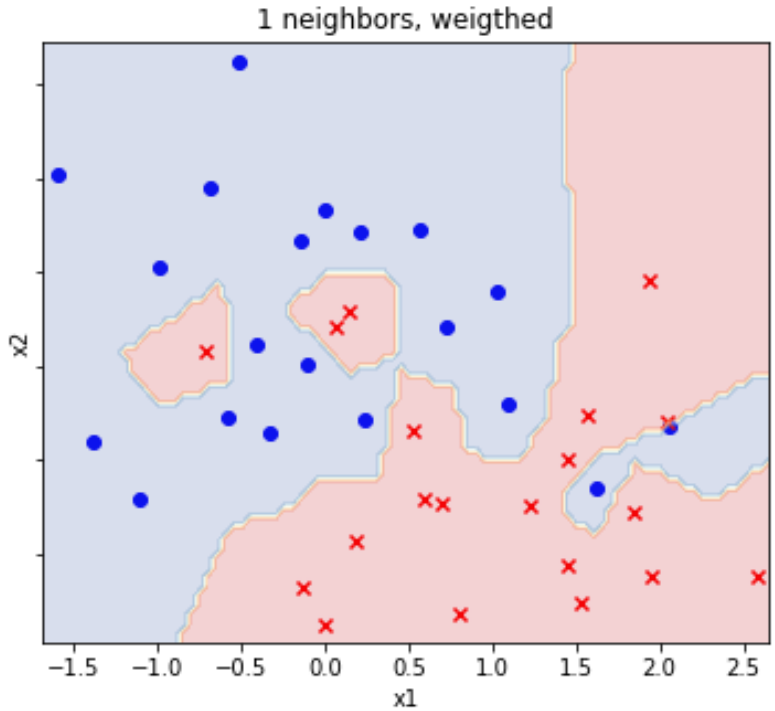

k = 8 neighbors
2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5


votes of the 8 NN
1.0
0.8
0.6
0.4
0.2
0.0
unit
weighted

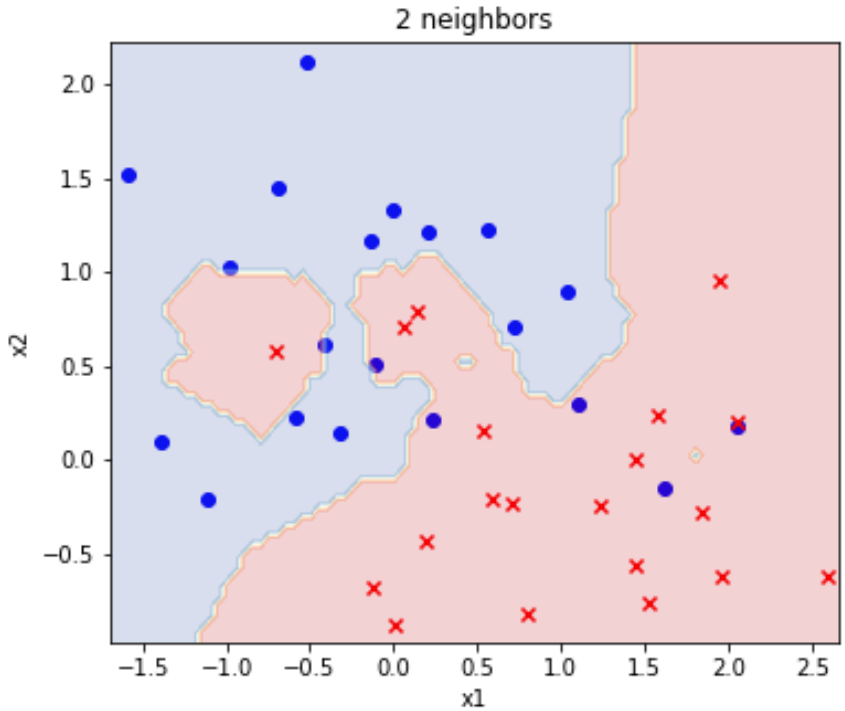
Index

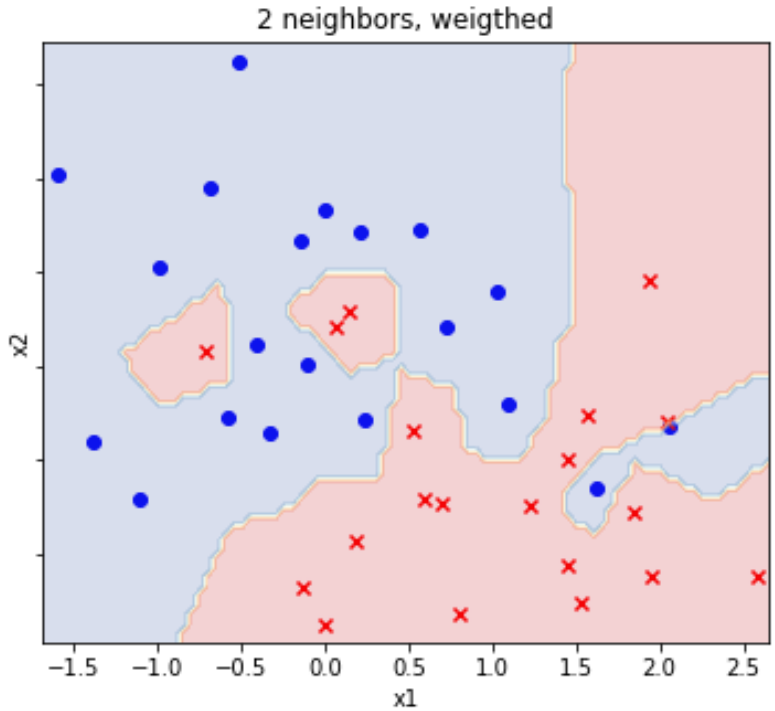
Classification regions



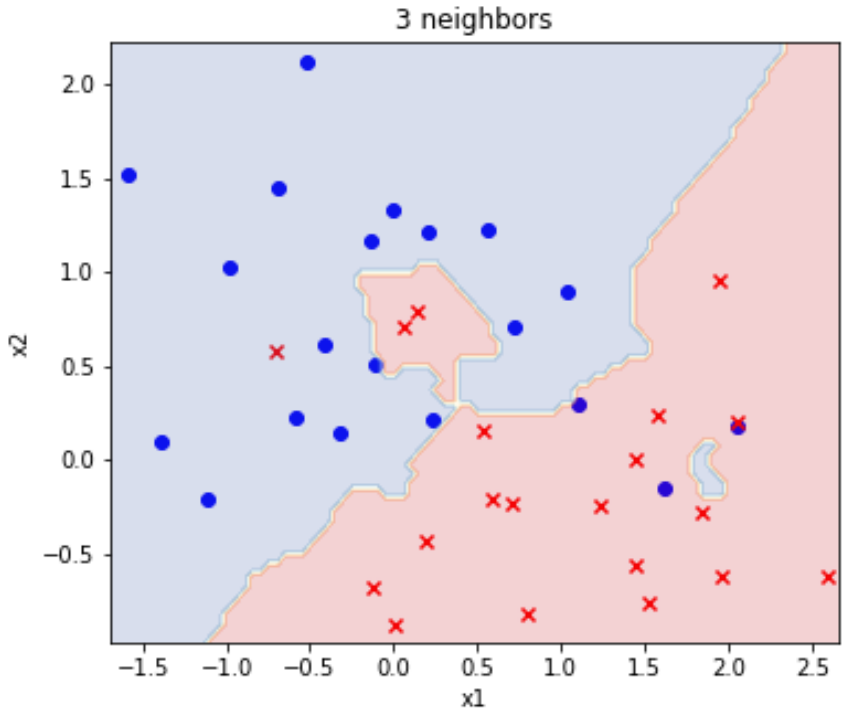


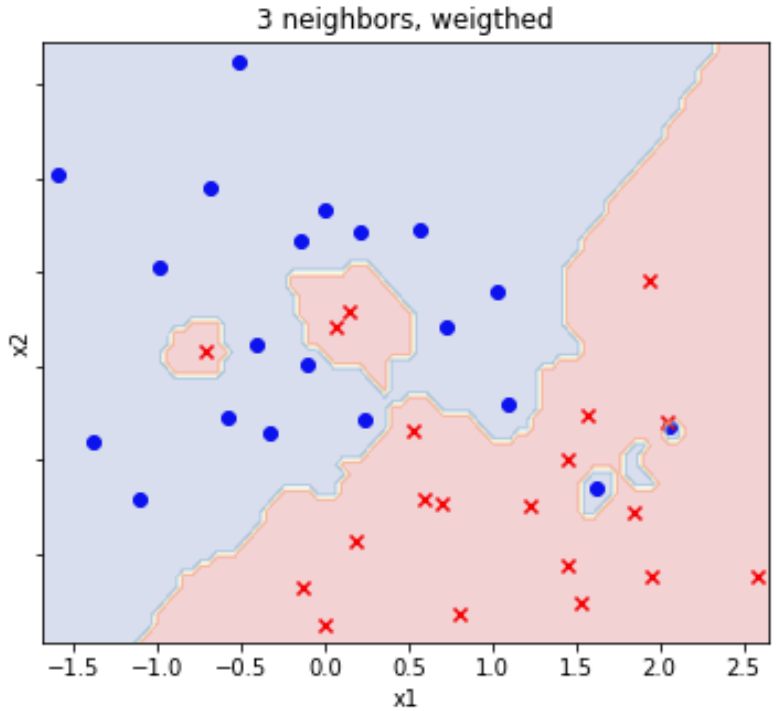
Classification regions



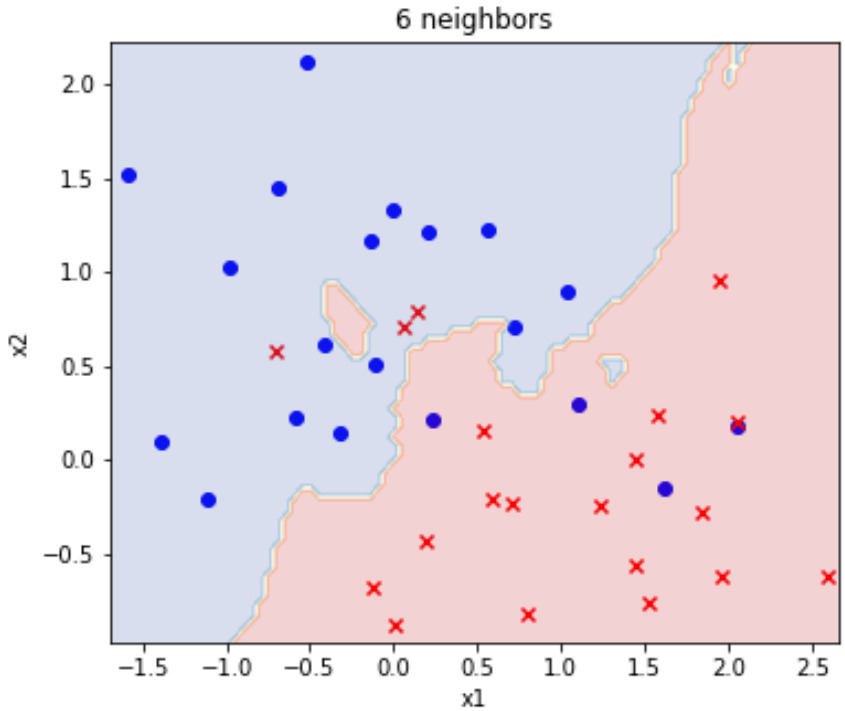


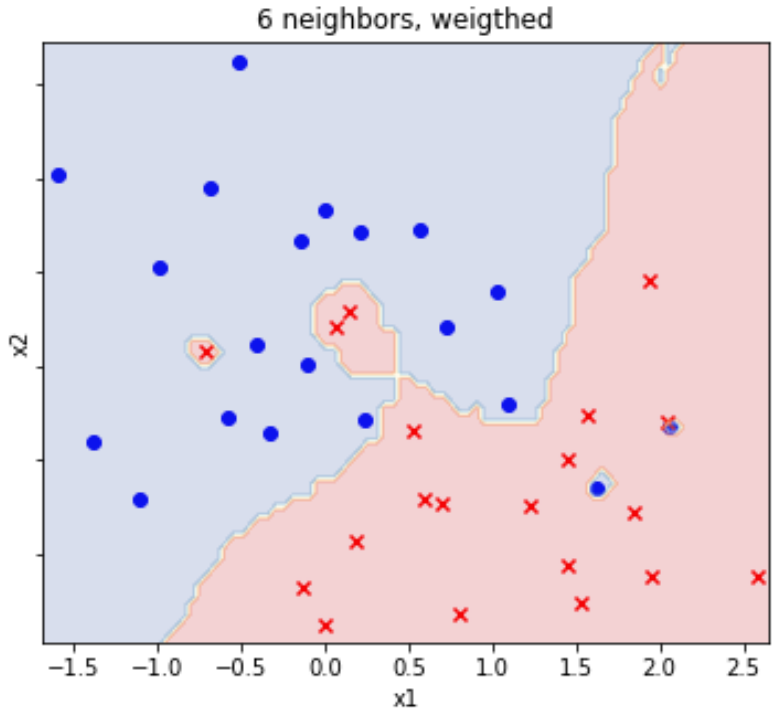
Classification regions





Classification regions





Index

Index

Stump classifier in 1D

A **decision tree** can be regarded as a hierarchical structure constructed through a recursive application of stump classifiers.

Stump classifier

Machine learning version of a binary threshold test on 1D observations

- Binary classifier: two ouput classes
- Defined by a threshold η

$$f_s(x) = x \begin{array}{c} D=1 \\ \gtrless \\ D=0 \end{array} \eta \rightarrow \text{ turns into } \rightarrow f_s(x) = \begin{cases} \text{True} & \text{if } x > \eta \\ \text{False} & \text{if } x \leq \eta \end{cases}$$

- Divides the input space in two, those observations for which the threshold test is True and those for which the threshold test is False.
- To complete the classifier, one needs to assign one of the classes to each outcome of the threshold test. This is made so that the classification accuracy is maximised.

Training a stump classifier in 1D

The algorithm to set the value of the threshold and the output class of each branch of the stump is the following

η, class_true, class_false = stump1d(X, $\mathbf{y}$)

1. Sort all the training observations along the value of the observation x.
2. There is a maximum of N possible ways of partitioning N observations with a single threshold on the value of η:
   - $(x_i + x_{i+1})/2$, $i = 1, \ldots, N-1$ and
   - either $\eta = -\infty$ or $\eta = \infty$, that is, all the observations are in the same class and thus there is no point in dividing the group.
3. Loop for all the N possibles values of η
   1. Assign a class to each side of the stump
      - class_true_n = mode of targets with $x > \eta_n$
      - class_false_n = mode of targets with $x \leq \eta_n$
   2. Evaluate the quality of the stump (for instance with the classification error or any other score)
4. Return the value of η and the corresponding branch class assignment that yielded the best value of the score

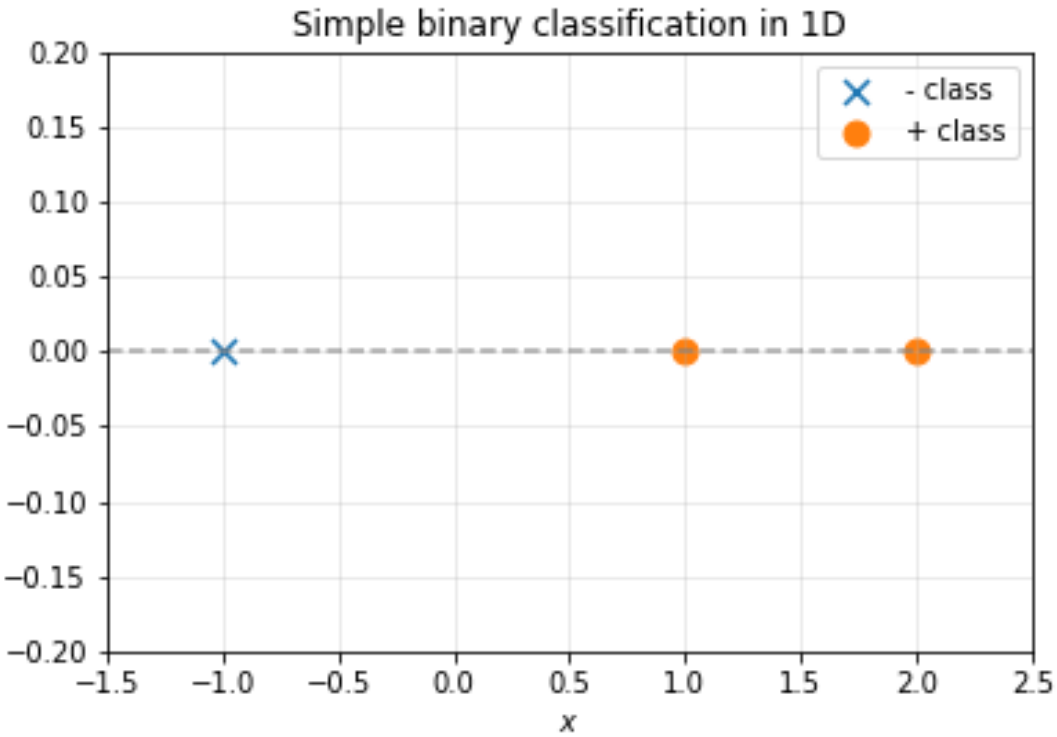
Evaluation of a stump classifier in 1D

$\hat{y}$ = evaluate_stump1d(x, η, class_true, class_false)

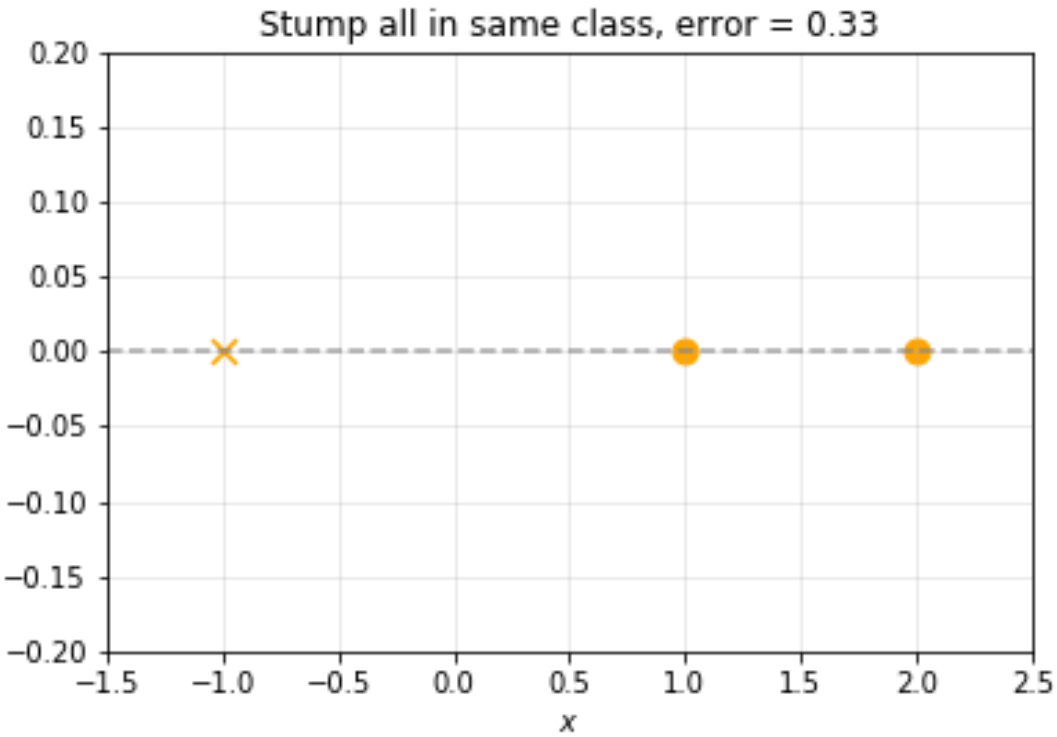
$$f_s(x) = \begin{cases} \text{class\_true} & \text{if } x > \eta \\ \text{class\_false} & \text{if } x \leq \eta \end{cases}$$

Return $\hat{y} = f_s(x)$

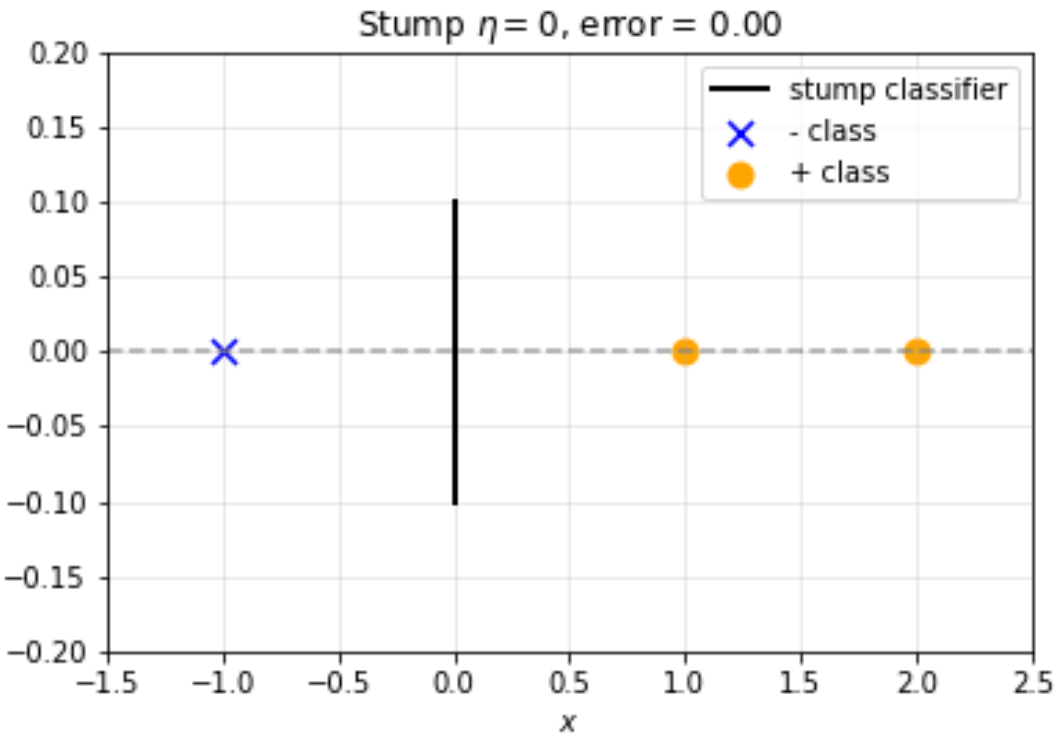
Example


Simple binary classification in 1D
- class
+ class
0.20
0.15
0.10
0.05
0.00
−0.05
−0.10
−0.15
−0.20
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0
2.5
x

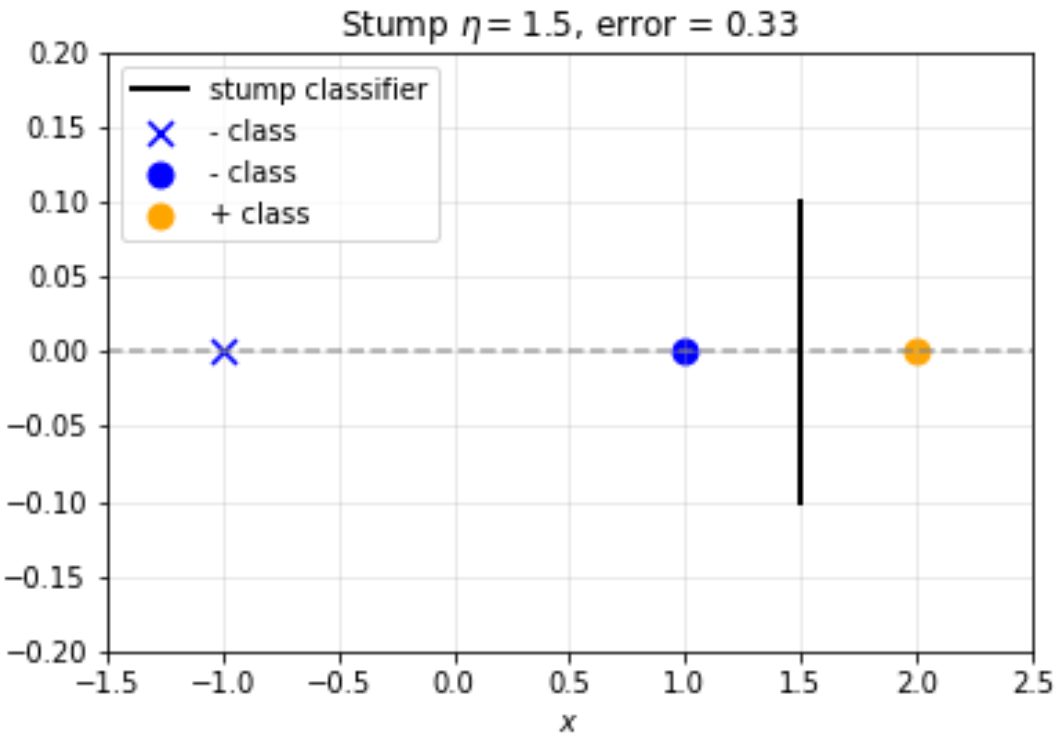
Example


Stump all in same class, error = 0.33
0.20
0.15
0.10
0.05
0.00
−0.05
−0.10
−0.15
−0.20
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0
2.5
x

Example

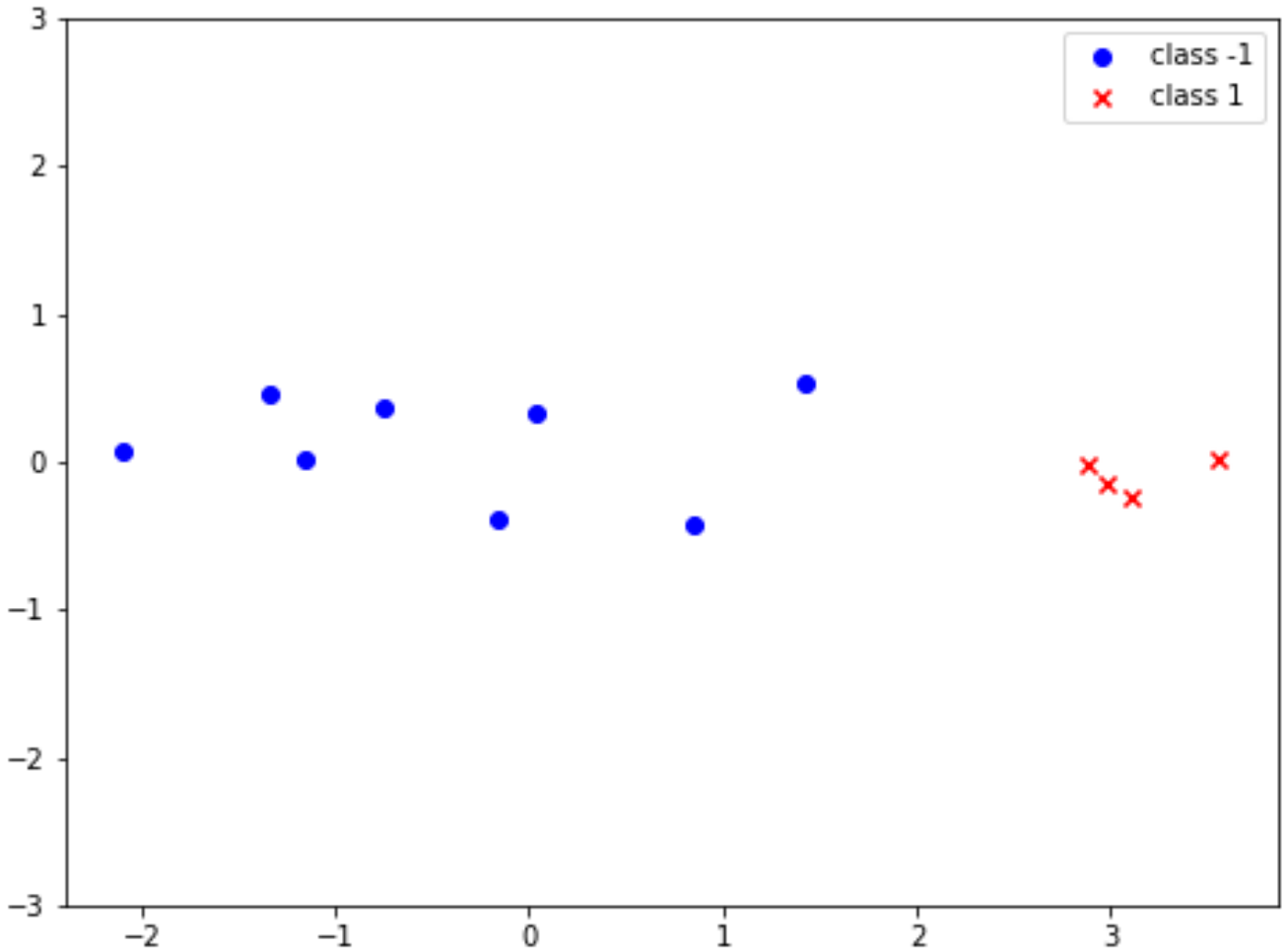

Stump $\eta = 0$, error = 0.00
stump classifier
- class
+ class
0.20
0.15
0.10
0.05
0.00
−0.05
−0.10
−0.15
−0.20
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0
2.5
x

Example


Stump $\eta = 1.5$, error = 0.33
stump classifier
- class
- class
+ class
0.20
0.15
0.10
0.05
0.00
−0.05
−0.10
−0.15
−0.20
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0
2.5
x

Stump classifiers in multidimensional problems

The stump classifier algorithm can be extended to observations with more than 1 dimension, $\mathbf{x} \in \mathbb{R}^d$ with $d > 1$.



Training stump classifiers in multidimensional problems

Basically one needs to run the `stump1d` algorithm within a nested loop that iterates over all the d dimensions of the observations.
After processing each dimension $j = 1, \ldots, d$, the `stump1d` will output a best stump for dimension j.
Once all the d dimensions are explored, the algorithm will return the stump corresponding to the dimension that achieved the best value of the score

j, η, class_true, class_false = stump(X, $\mathbf{y}$)

1. Loop for all the dimensions of the observations, in this case, the number of columns of data matrix X.
   `For` $j = 1, \ldots, d$
   - Z is an N dimensional array with the j-th column of X
   - η_j`, class_true_j, class_false_j = stump1d(`Z`, y)`
   - store in a variable s_j the score yielded by (Z,$\mathbf{y}$) in the stump defined with η_j`, class_true_j, class_false_j`
2. Find the best score out of all the iterations
3. Return the dimension j_* that achieved the best score and the corresponding stump defined with η_{j_*} and the corresponding branch class assignment

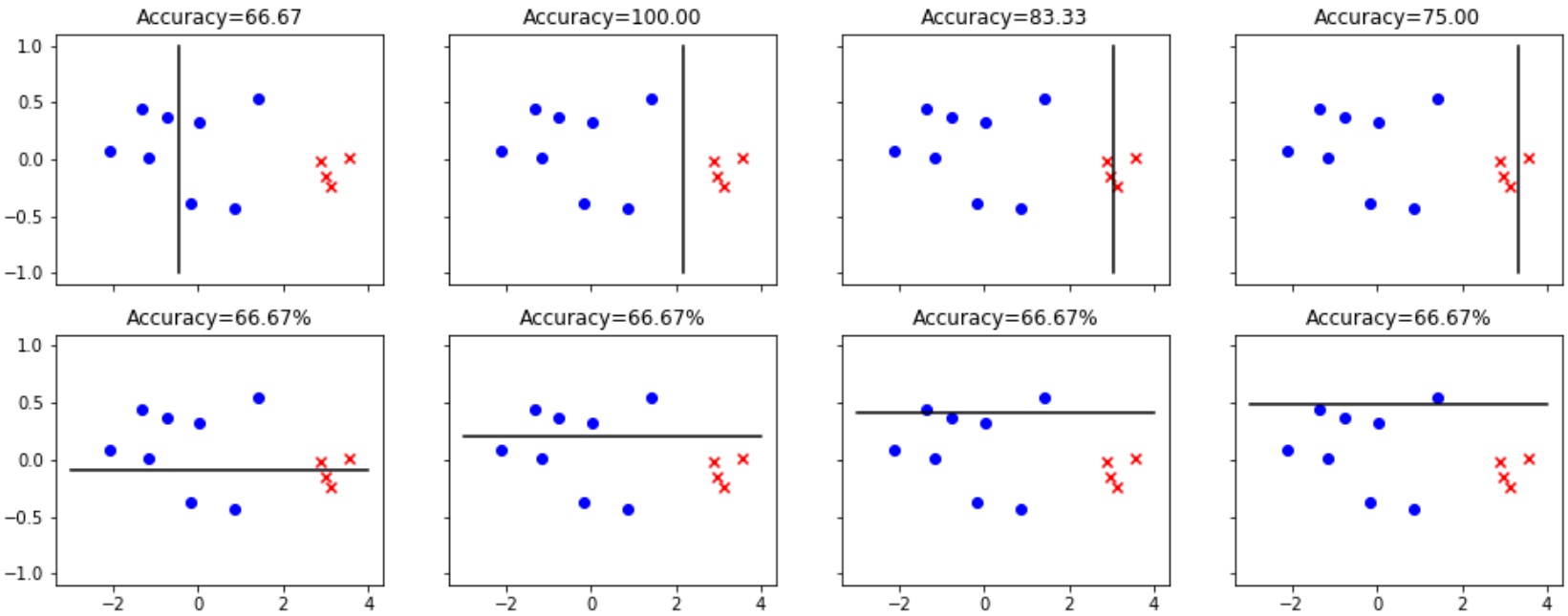
Evaluation of a stump classifier with multidimensional data

$\hat{y}$ = evaluate_stump($\mathbf{x}$, j, η, class_true, class_false)

$$f_s(\mathbf{x}) = \begin{cases} \text{class\_true} & \text{if } x_j > \eta \\ \text{class\_false} & \text{if } x_j \leq \eta \end{cases}$$

Return $\hat{y} = f_s(\mathbf{x})$

Example


Accuracy=66.67
Accuracy=100.00
Accuracy=83.33
Accuracy=75.00
Accuracy=66.67%
Accuracy=66.67%
Accuracy=66.67%
Accuracy=66.67%
1.0
0.5
0.0
−0.5
−1.0
−2
0
2
4

Index

Tree construction

Tree construction

1. Start with a **root node**, s_0, holding the whole training set
2. Divide s_0 into two **branch nodes** (s_1 and s_2) with the algorithm `stump`.
3. Place s_1 and s_2 in a list of branch nodes to explore $\mathcal{B}$. Notice each branch node also receives a subset of the training set: s_1 receives the observations that the stump in s_0 classifies as True while s_2 receives the observations that s_0 classifies as False.
4. Apply the following **recursion** to each node s_k in $\mathcal{B}$. Explore s_k with the algorithm `stump` and the training subset received from its parent:
   - If s_k is **homogeneous** (the stump does not split the observations) s_k becomes a **leaf node**. Then s_k gets a label (the majority class among all the training observations that arrived to it) to classify the test observations that end in s_k.
   - If s_k gets divided by the stump in two branches s_j and s_l:
     - Place s_j and s_l in $\mathcal{B}$
     - Assign to s_j and to s_l the corresponding subsets of the training observations that had arrived to s_k
   - Remove s_k from $\mathcal{B}$
5. Run the recursion until $\mathcal{B}$ is empty

Impurity scores for classification

Each node of the tree is explored with the `stump` algorithm. The stump needs a score to assess the **quality** of the split. Common scores for classification problems are:

- Classification error rate
- **Gini Index**. In a problem with K classes, we define p_k as the fraction of observations in the node that belong to class t_k

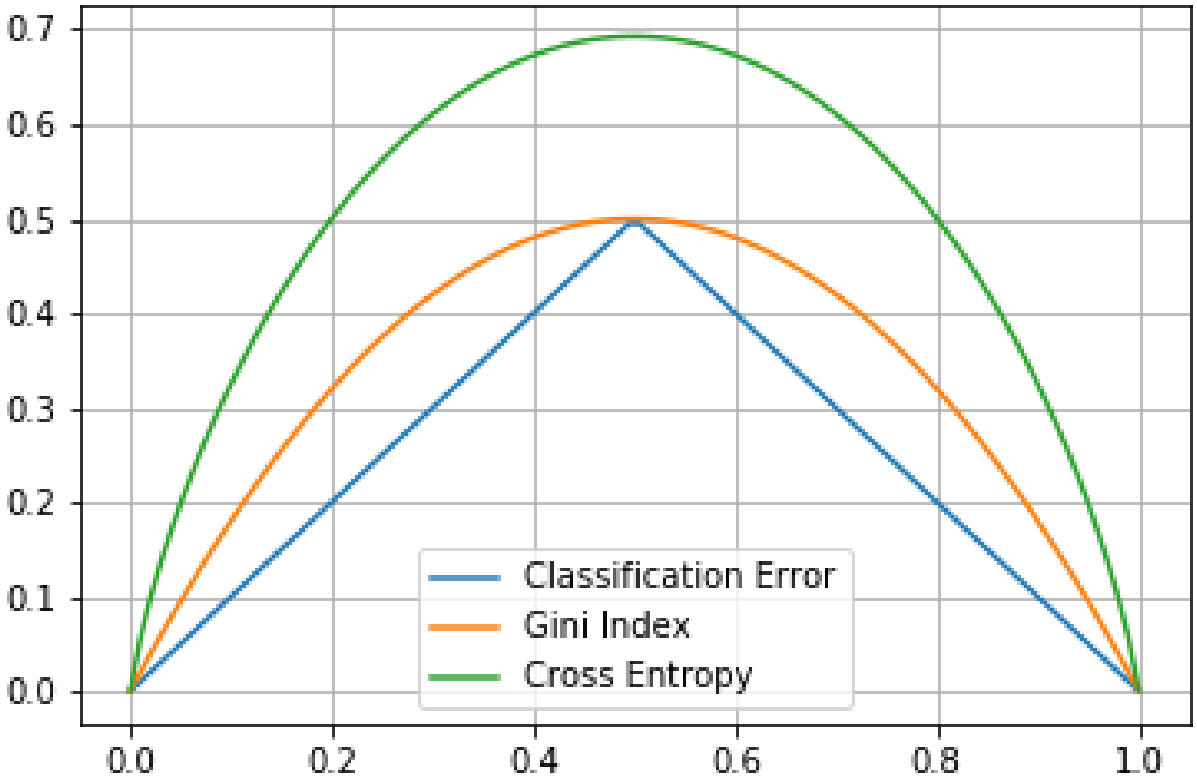
$$\text{Gini} = \sum_{k=1}^{K} p_k(1 - p_k)$$

- **Cross entropy**

$$-\sum_{k=1}^{K} p_k \log p_k$$

Intuitively the Gini index or the cross entropy seem a better choice over the classification error since in cases in which two splits present a same classification error, they favor the split with a purer descendent.

Impurity scores for classification with 2 classes



Tree evaluation

Tree evaluation

1. The evaluation of each observation starts with the stump in the root node s_0. The outcome of this test drives each observation through one of the two nodes (s_1 or s_2) that descend from s_0
2. The observation recursively travels down the hierarchy of the tree, guided by the stumps in the branches
3. The observation finishes its travel when it arrives to a leaf node. It then gets classified with the class of the leaf node.

Index

Controlling the growth of the tree

- The algorithm to grow the tree ends when $\mathcal{B}$ is empty. This means all the training set observations has ended in a leaf node. In a worse case the tree will end up with many leaves classifying a single training observation, what surely will lead to a poor generalization capability.
- Note also that the quality of the stump in the branch nodes depends on the size of the training subset that arrives to them. So as the tree grows deeper the quality of the stumps worsens.
- To overcome this situation one can introduce some external rules to control the growth of the tree. These rules are encoded into **hyperparameters** that are checked along the corresponding steps of the growing tree algorithm.

Hyperparameters for growing trees

- criterion for the quality of the stumps: "gini" or "entropy"
- maximum depth of the tree
- minimum number of samples required to split an internal node. If s_k receives less samples it becomes a leaf
- minimum number of samples required to be at a leaf node. If a branch receives less samples, then the parent becomes a leaf
- number of features to consider when looking for the best split. This means the algorithm `stump` does not explore all the columns of X, just a random subset in each split
- maximum number of leafs. Explore $\mathcal{B}$ in a bf best-first fashion until this limit is reached. Then all the nodes in $\mathcal{B}$ become leaves
- minimum decrease in the impurity score to split a node. If the weighted impurities of the two branches of a node s_k do not improve this limit then s_k becomes a leaf.

Parameters and hyperparameters

During the training of a machine learning model such as a decision tree for classification on needs to find values for two kinds of parameters:

Fittable parameters

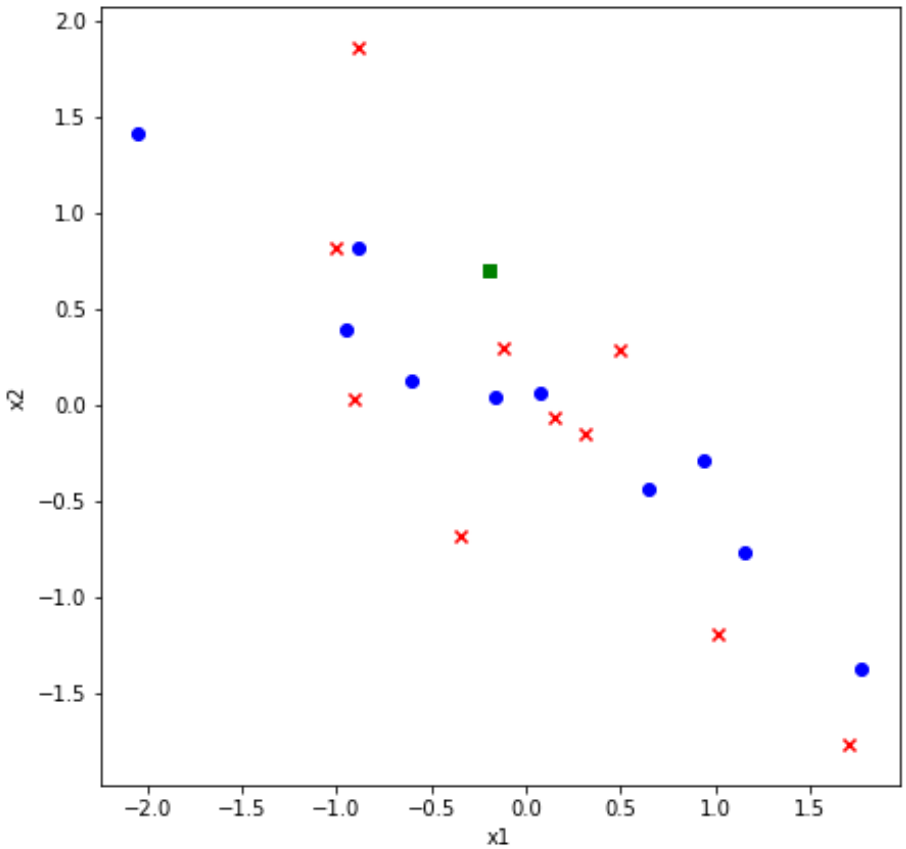
The parameters that define each stump in the tree: the feature j and the threshold η. Their values are adjusted during the optimization that involves the fitting of the model

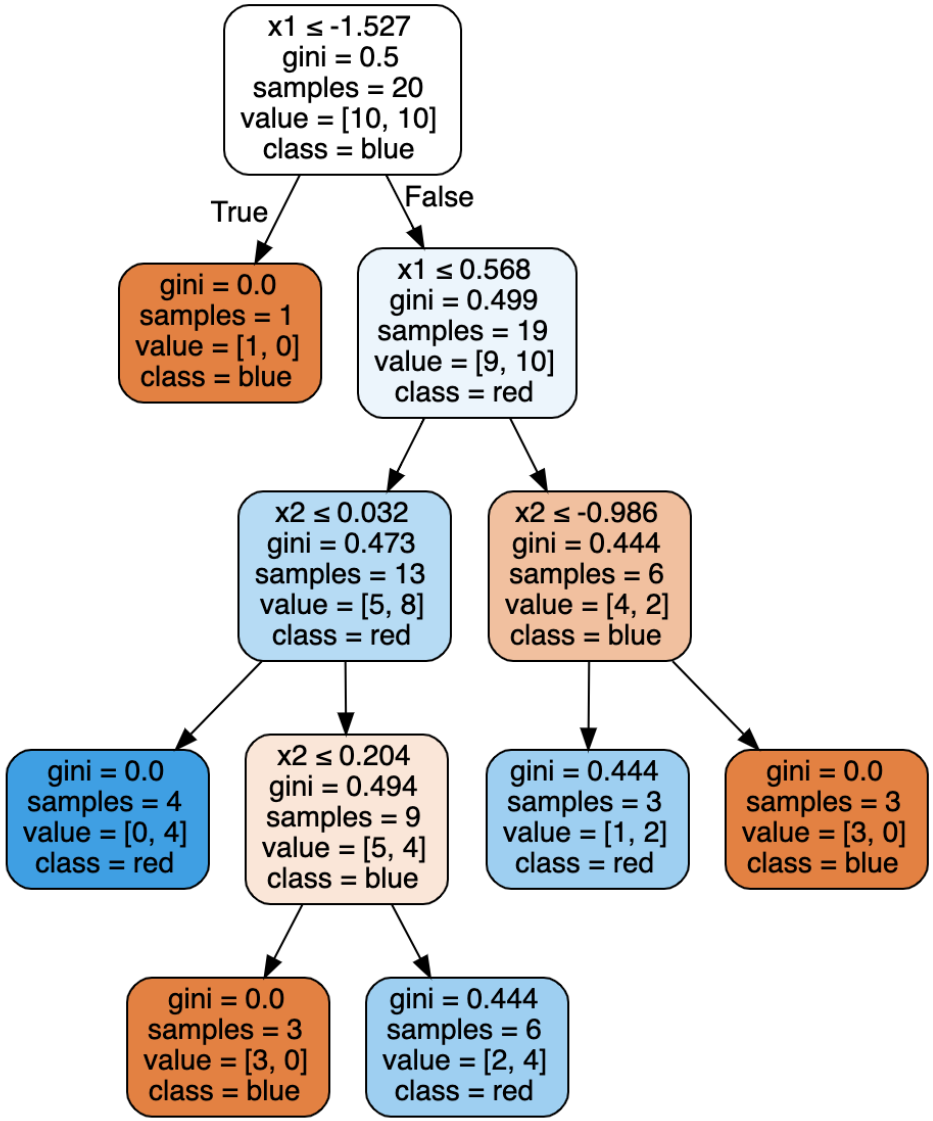
Hyperparameters

They are set during the instantiation of the model. Their role is to control the optimization within the fitting of the model. Their values are adjusted using **prior domain knowledge**. This way we control the generalization capability of the model.
Examples of hyperparameters are k in kNN or the maximum deep of the tree in a decision tree.

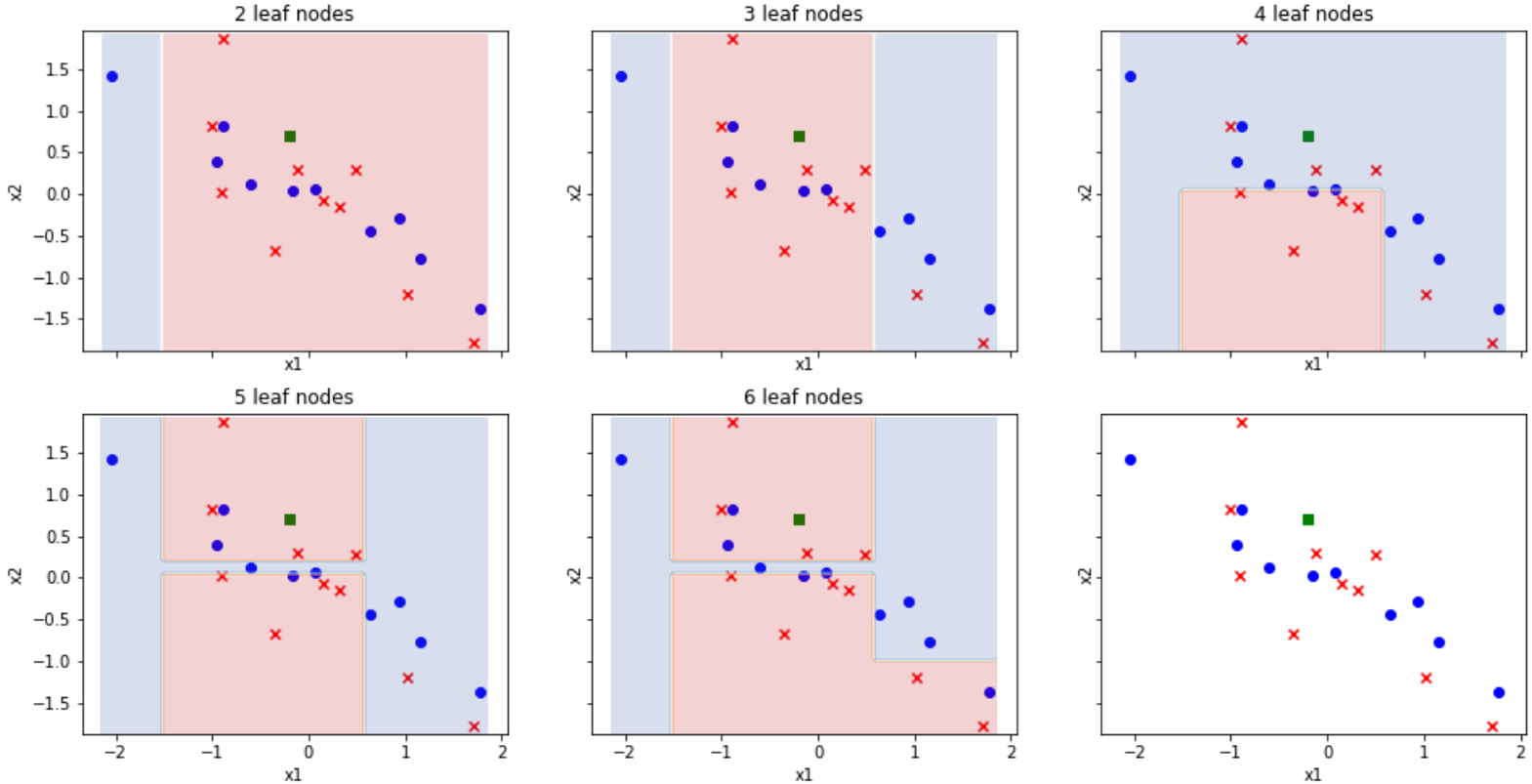
Index

Classification with 2 classes in 2D





Decision trees classification regions



Index

Index

Ensemble methods

A common way of developing robust machine learning models is to learn an **ensemble** of models with a same training set but introducing a small amount of **diversity** in each member of the ensemble:

- **subsampling** the common training set to get a slightly different set for every learner
- using different hyperparameters (maybe chosen at random) for every learner.

Weak Learners

Every member of the ensemble is called **weak learner** since you don't need its using a very accurate learning algorithm. In binary classification there are guarantees that as long as every weak learner is able to perform slightly above 50% accuracy the ensemble can converge to best possible performance ranges.

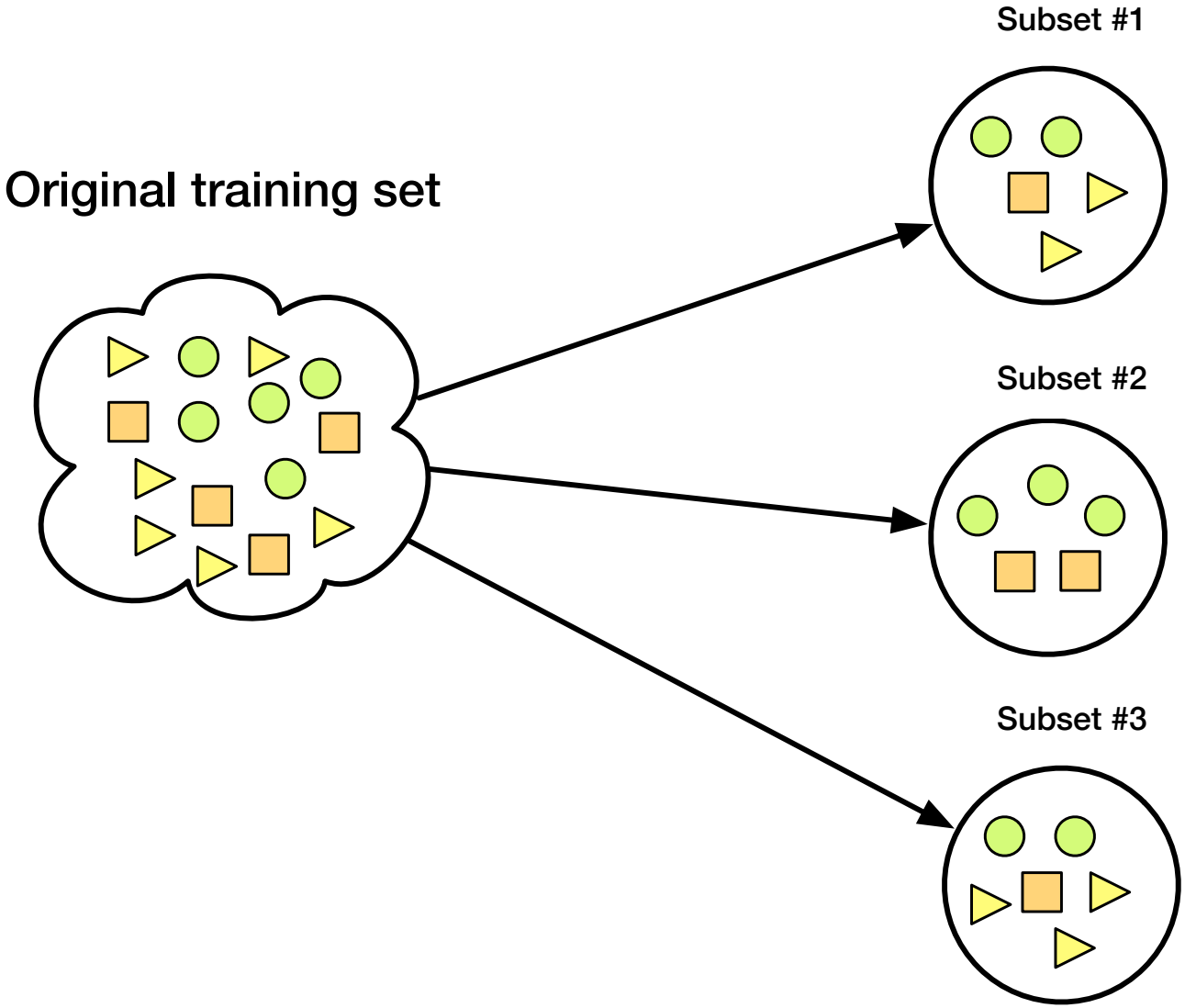
Once the ensemble is built, the prediction for each test observation is made by combining all the weak learners individual predictions

- Taking the average of the individual predictions
- Taking the mode of the individual predictions
- Weighting each individual learner prediction by a coefficient that captures the confidence on its predictions.

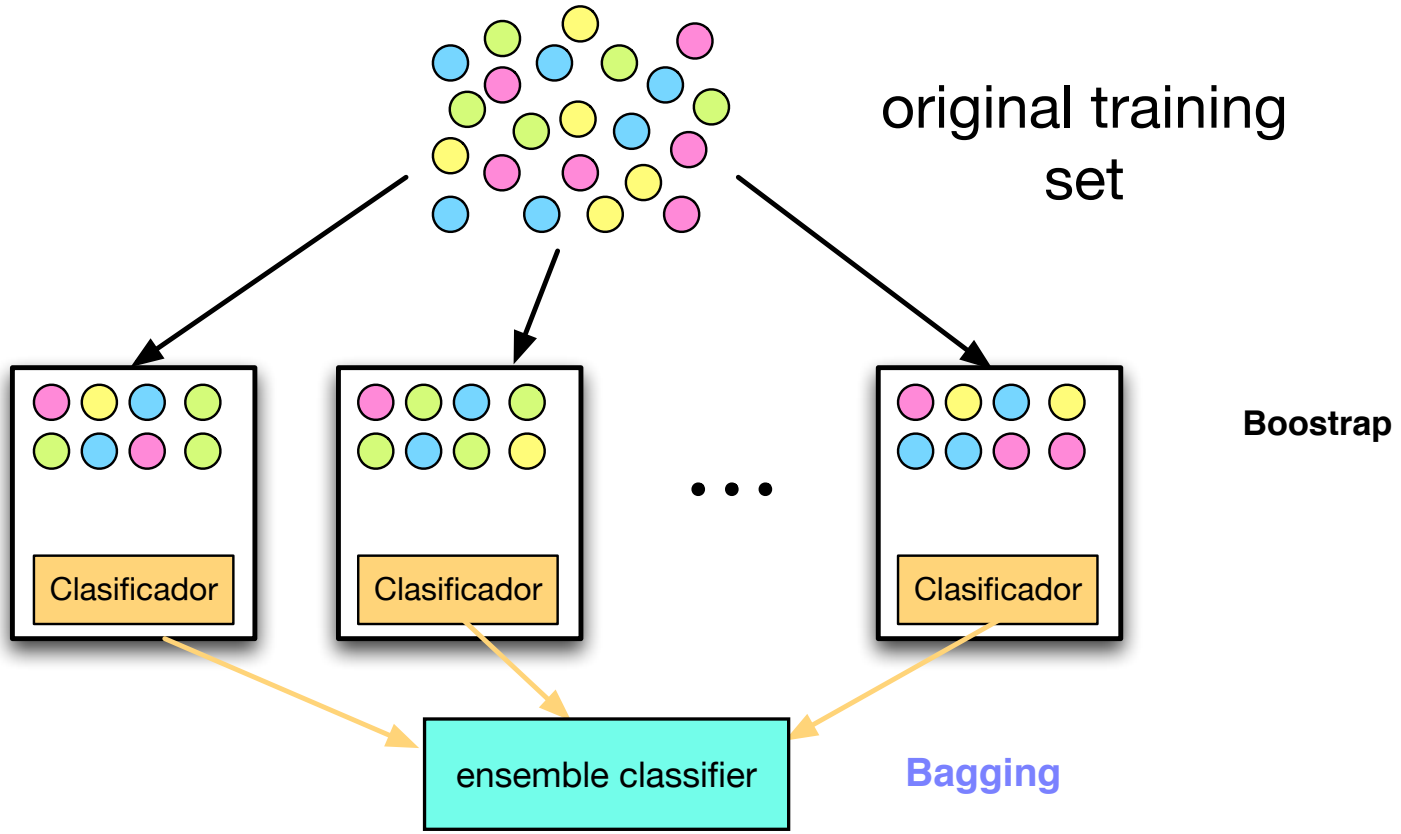
Bagging

- Every weak learner is fitted with a same algorithm but with a different training set that is drawn at random from the original training set (**bootstrap**)
- Weak learners can be fitted in parallel
- The output of the ensemble is the average of the outputs of the weak learners
- Examples: Random Forest

Bootstrap sketch



Bagging sketch



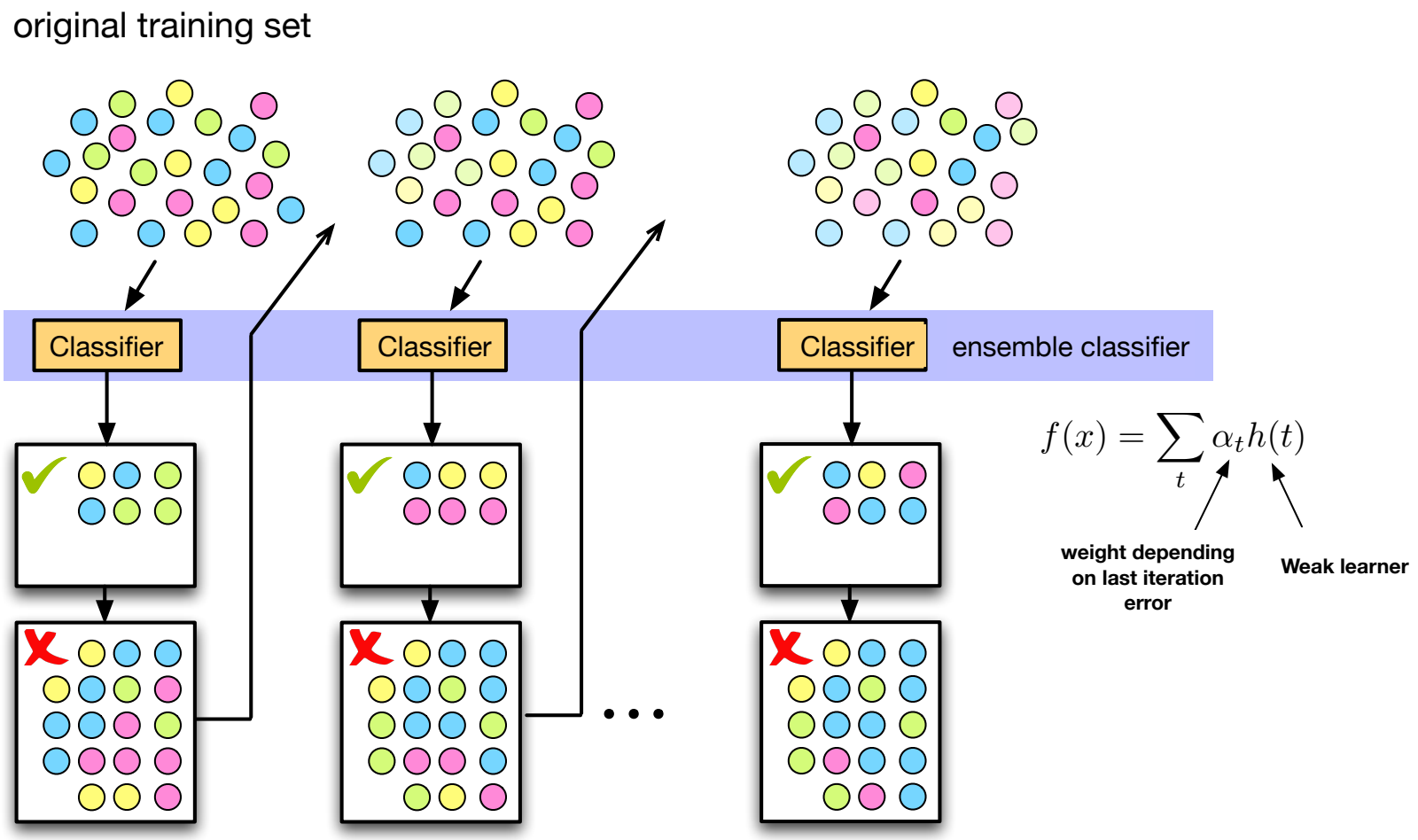
$$f(x) = \frac{1}{B}\sum_{b=1}^{B} f_b(x)$$

weak learner

Boosting

- Weak learners are trained sequentially using a same algorithm but with slightly different training sets.
- The first weak learner is fitted with the original training set
- The second weak learner receives a training set constructed by resampling the original set in a way that gives more weight to those observations missclassified by the first learner.
- Subsequent weak learners receive a training set constructed by resampling the original set in a way that gives more weight to those observations missclassified by the ensemble formed by the previous weak learners.
- Every weak learner added to the ensemble therefore aims at correcting the errors of its predecessors
- The output of the ensemble is the weighted average of the outputs of the weak learners. The weighting coefficients are proportional of the accuracy of each weak learner.

Boosting sketch



Committee of experts

- This ensemble is not formed by weak learners, but for more sophisticate and diverse algorithms.
- Each algorithm specializes in learning a part of the problem.
- A machine learning model called **gate** is fitted to decide which of the expert would classify each test observation using as information the performance of the different experts on the training set.

Index

Random Forest fitting

Random Forest form an ensemble by growing B decision trees.
The algorithm basically consists in a loop that runs an iteration for each tree in the forest (B iterations). The algorithm receives as input a training set with N observations in d dimensions, with their corresponding targets, and a forest size B.

Random Forest Algorithm

For $b = 1, \ldots, B$:

1. Choose at random a subset of N_b observations of the training set
2. Learn tree T_b with this training set of size N_b but with this modification in the standard tree growing algorithm: Before optimizing every stump for every branch node, choose at random m_b features from the available d. This way, each node only access a subset of the input features to optimize its stump, what intoduces further diversity in the growing of the forest
3. Inference. Each tree outputs a target for each test observation. The final target output by the forest is the most voted class

Index

Random Forest for classification regions