Master's Degree in Mechatronic Engineering

Politecnico di Torino

Master's Degree Thesis

UNSUPERVISED MACHINE LEARNING ALGORITHMS FOR EDGE NOVELTY DETECTION

Supervisors

Prof. Marcello CHIABERGE

Dott. Umberto ALBERTIN

Dott. Gianluca DARA

Candidate

Ariel PRIARONE

03 2024

Acknowledgements

I would like to thank the PoliTO Interdepartmental Centre for Service Robotics (PIC4Ser) for giving me the opportunity to work on this project. The guidance and infrastructure provided by the centre have been invaluable during the development of this work.

To my parents, who have given me everything.
Thank you for always making me believe that anything is possible.
Ai miei genitori, che mi hanno dato tutto.
Grazie per avermi sempre fatto credere che tutto sia possibile.

Ariel

Table of Contents

List of Tables

List of Figures

Chapter 1

Machine Learning

Before diving into the description of the unsupervised algorithms used for the development of this thesis work presented in **??**, this chapter aims to be an introduction of *Machine Learning* (ML) in general.

An early but useful definition of Machine Learning was given by Arthur Samuel in 1959: "*Machine learning is the field of study that gives computers the ability to learn without being explicitly programmed.*" A more recent definition is the following, from Tom Mitchell: "*A computer program is said to learn from experience E with respect to some task T and some performance measure P, if its performance on T, as measured by P, improves with experience E.*" [1, p. 4]

So, in general, the ingredients of ML are:

- some data linked to some task
- a task to be performed
- an algorithm that learns how to perform the task on specific data

The data are usually preprocessed before giving them to the algorithm. The processed data are called *features*. This is a generic term that refers to the information content of the data. For example, if the data are recordings of temperatures over time, the features could be the mean, the standard deviation, the minimum, and the maximum of the temperature or, in some cases if the algorithm is able to learn directly from them, the raw data themself.

The tasks can be divided into main categories:

- regression: the algorithm is trained to measure the relation between the value of output variables and corresponding values of other input variables;
- classification: the algorithm is trained to assign a label to a new instance, based on the training dataset of labelled instances;

- clustering: the algorithm is trained to group similar instances into clusters.
- anomaly detection: the algorithm is trained to identify instances that are different from known previous instances.

1.1 Regression

1.1.1 Least Squares

Let's consider a set of m observations of a variable $y \in \mathbb{R}^{n_y}$ (output features) that depends on a variable $x \in \mathbb{R}^{n_x}$ (input features) and a set of $n_f \cdot n_y$ parameters $\theta \in \mathbb{R}^{n_f \times n_y}$.

Suppose to know that the output features are linked to the input features with n_f functions linear in the parameters θ, so that:

$$\begin{bmatrix} y_1 & y_2 & \dots & y_{n_y} \end{bmatrix} =$$

$$\begin{bmatrix} f_1(x_1, \dots, x_{n_x}) & f_2(x_1, \dots, x_{n_x}) & \dots & f_{n_f}(x_1, \dots, x_{n_x}) \end{bmatrix} \cdot \begin{bmatrix} \theta_{1,1} & \dots & \theta_{1,n_y} \\ \theta_{2,1} & \dots & \theta_{2,n_y} \\ \vdots & \ddots & \vdots \\ \theta_{n_f,1} & \dots & \theta_{n_f,n_y} \end{bmatrix}$$

Where all the f_i are any known functions, y_i and x_i are known data and $\theta_{i,j}$ are the parameters to be found.

Considering the m observations, the previous equation can be extended as:

$$\begin{bmatrix} y_{1,1} & y_{1,2} & \dots & y_{1,n_y} \\ y_{2,1} & y_{2,2} & \dots & y_{2,n_y} \\ \vdots & \ddots & \vdots & \\ y_{m,1} & y_{m,2} & \dots & y_{m,n_y} \end{bmatrix} =$$

$$\begin{bmatrix} f_1(x_{1,1}, \dots, x_{1,n_x}) & \dots & f_{n_f}(x_{1,1}, \dots, x_{1,n_x}) \\ f_1(x_{2,1}, \dots, x_{2,n_x}) & \dots & f_{n_f}(x_{2,1}, \dots, x_{2,n_x}) \\ \vdots & \ddots & \vdots \\ f_1(x_{m,1}, \dots, x_{m,n_x}) & \dots & f_{n_f}(x_{m,1}, \dots, x_{m,n_x}) \end{bmatrix} \cdot \begin{bmatrix} \theta_{1,1} & \dots & \theta_{1,n_y} \\ \theta_{2,1} & \dots & \theta_{2,n_y} \\ \vdots & \ddots & \vdots \\ \theta_{n_f,1} & \dots & \theta_{n_f,n_y} \end{bmatrix}$$

Rewriting the previous equation in a more compact form:

$$\begin{bmatrix} \boldsymbol{y}_1 \\ \boldsymbol{y}_2 \\ \vdots \\ \boldsymbol{y}_m \end{bmatrix} = \begin{bmatrix} f_1(\boldsymbol{x}_1) & f_2(\boldsymbol{x}_1) & \dots & f_{n_f}(\boldsymbol{x}_1) \\ f_1(\boldsymbol{x}_2) & f_2(\boldsymbol{x}_2) & \dots & f_{n_f}(\boldsymbol{x}_2) \\ \vdots & \ddots & \vdots & \\ f_1(\boldsymbol{x}_m) & f_2(\boldsymbol{x}_m) & \dots & f_{n_f}(\boldsymbol{x}_m) \end{bmatrix} \cdot \begin{bmatrix} \theta_{1,1} & \dots & \theta_{1,n_y} \\ \theta_{2,1} & \dots & \theta_{2,n_y} \\ \vdots & \ddots & \vdots \\ \theta_{n_f,1} & \dots & \theta_{n_f,n_y} \end{bmatrix} \tag{1.1}$$

That, in the most compact form, becomes:

$$\boldsymbol{Y} = \boldsymbol{\Phi}(\boldsymbol{X}) \cdot \boldsymbol{\Theta} \tag{1.2}$$

In close form, there is a solution $\boldsymbol{\Theta_{LS}}$, for estimating the parameters that minimize the error between the estimated output $\boldsymbol{Y_{LS}} = \boldsymbol{\Phi(X)\Theta_{LS}}$ and the real output $\boldsymbol{Y}$, that is known. Let's see, in an intuitive way:

$$\begin{aligned}
\boldsymbol{Y} &= \boldsymbol{\Phi}(\boldsymbol{X}) \cdot \boldsymbol{\Theta_{LS}} && (1.3)\\
\boldsymbol{\Phi}(\boldsymbol{X})^T\boldsymbol{Y} &= \underbrace{\boldsymbol{\Phi}(\boldsymbol{X})^T\boldsymbol{\Phi}(\boldsymbol{X})}_{\text{square}} \cdot \boldsymbol{\Theta_{LS}} && (1.4)\\
(\boldsymbol{\Phi}(\boldsymbol{X})^T\boldsymbol{\Phi}(\boldsymbol{X}))^{-1}\boldsymbol{\Phi}(\boldsymbol{X})^T\boldsymbol{Y} &= \boldsymbol{\Theta_{LS}} && (1.5)\\
\text{pinv}(\boldsymbol{\Phi}(\boldsymbol{X}))\boldsymbol{Y} &= \boldsymbol{\Theta_{LS}} && (1.6)
\end{aligned}$$

In fact, it is known that $\boldsymbol{\Theta_{LS}} = \text{pinv}(\boldsymbol{\Phi}(\boldsymbol{X}))\boldsymbol{Y}$ is the solution of the following minimization problem:

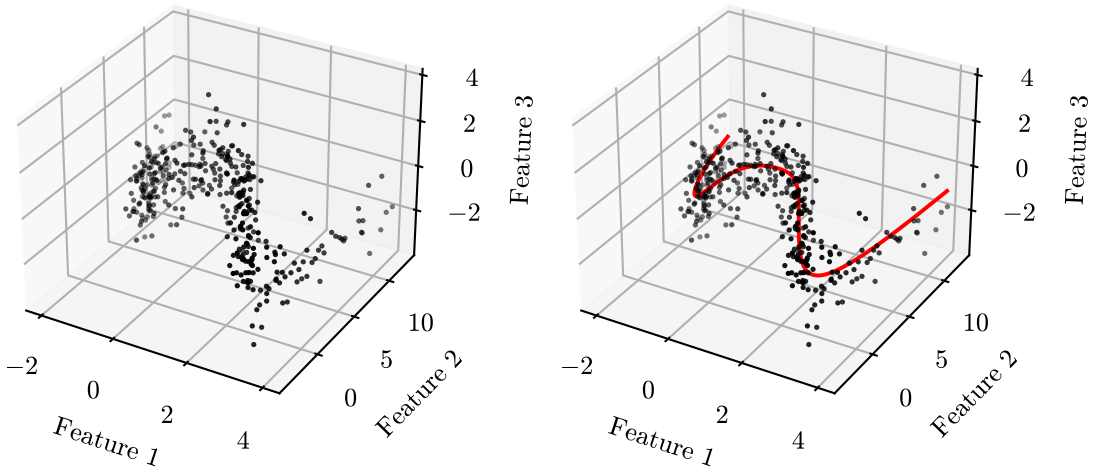
$$\boldsymbol{\Theta_{LS}} = \arg \min_{\boldsymbol{\Theta} \in \mathbb{R}^{n_f \times n_y}} \|\boldsymbol{Y} - \boldsymbol{\Phi}(\boldsymbol{X})\boldsymbol{\Theta}\|_2^2 \tag{1.7}$$

That is why this method is called *Least Squares* (LS). It is proven that if the data $\boldsymbol{Y}$ affected my white noise, and the data $\boldsymbol{X}$ are known precisely, the solution converges to the real parameters $\boldsymbol{\Theta}_{\text{true}}$ when the number of observations m goes to infinity.

$$\lim_{m \to \infty} \boldsymbol{\Theta_{LS}} = \boldsymbol{\Theta}_{\text{true}} \tag{1.8}$$

Is this considered machine learning? Yes, even being just a simple implementation of linear algebra, once programmed in a computer, it qualifies as the (simplest) machine learning algorithm because fitting new data does not require any human intervention. Let's see an example. Suppose to have 400 data points, shown in **figure 1.1a**, of the variable x, y_1 and y_2 sampled with noise, that we call Feature 1, Feature 2 and Feature 3, respectively. Suppose that it is known that the output features are linked to the input feature with a linear combination of the functions e^x, x^3, $\cos(x)$, $\sin(x)$ and $\cos^3(x)$, but the parameters θ are unknown:

$$\begin{aligned}
y_1 &= \theta_{1,1}e^x + \theta_{2,1}x^3 + \theta_{3,1}\cos(x) + \theta_{4,1}\sin(x) + \theta_{5,1}\cos^3(x) && (1.9)\\
y_2 &= \theta_{1,2}e^x + \theta_{2,2}x^3 + \theta_{3,2}\cos(x) + \theta_{4,2}\sin(x) + \theta_{5,2}\cos^3(x) && (1.10)
\end{aligned}$$



(a) *400 data points* **(b)** *data points and the fitted curve*

Figure 1.1: *Least square regression example*

rearranging in matrix form:

$$\underbrace{\begin{bmatrix} y_{1,1} & y_{1,2} \\ y_{2,1} & y_{2,2} \\ \vdots & \vdots \\ y_{m,1} & y_{m,2} \end{bmatrix}}_{\boldsymbol{Y}} = \underbrace{\begin{bmatrix} e^{x_1} & x_1^3 & \cos(x_1) & \sin(x_1) & \cos^3(x_1) \\ e^{x_2} & x_2^3 & \cos(x_2) & \sin(x_2) & \cos^3(x_2) \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ e^{x_m} & x_m^3 & \cos(x_m) & \sin(x_m) & \cos^3(x_m) \end{bmatrix}}_{\boldsymbol{\Phi}(\boldsymbol{X})} \cdot \underbrace{\begin{bmatrix} \theta_{1,1} & \theta_{1,2} \\ \theta_{2,1} & \theta_{2,2} \\ \theta_{3,1} & \theta_{3,2} \\ \theta_{4,1} & \theta_{4,2} \\ \theta_{5,1} & \theta_{5,2} \end{bmatrix}}_{\boldsymbol{\Theta}} \tag{1.11}$$

applying the LS solution from **equation 1.6**, we obtain:

$$\boldsymbol{\Theta_{LS}} = \text{pinv}(\boldsymbol{\Phi}(\boldsymbol{X}))\boldsymbol{Y} = \begin{bmatrix} +1.997 & -0.004 \\ -1.498 & +0.003 \\ +1.332 & -0.018 \\ -0.005 & +0.999 \\ -0.032 & +1.035 \end{bmatrix}$$

that is quite close to the real parameters used to generate the data:

$$\boldsymbol{\Theta}_{\text{true}} = \begin{bmatrix} +2.0 & +0 \\ -1.5 & +0 \\ +1.3 & +0 \\ +0.0 & +1 \\ +0.0 & +1 \end{bmatrix}$$

Using the estimated parameters, it is possible to estimate the output features for new input features, the regression line is shown in **figure 1.1b**.

Applicability

This is an elegant closed-form solution for a regression problem, however, it has some limitations:

- if the noise is not white, or it is present also in the input features, the solution is not guaranteed to converge to the real parameters;
- if there are nonlinearities in the parameters (for example $\sin(\theta_{1,1}x)$), the solution is not applicable;

1.1.2 Gradient Descent GD

To overcome these limitations, another way to estimate the parameters is to use an iterative algorithm that minimizes a cost function over the parameters space. The iterations aim to update the parameters in the direction of the steepest descent of the cost function. This can be done even with nonlinearities in the data, and even if the noise is not white, but has the drawback of the risk of getting stuck in a local minimum of the cost function, starting from a random initialization. Another limitation is the fact that a learning rate η has to be defined, that is a parameter that defines how much the parameters are updated at each iteration. If the learning rate is too small, the algorithm will take a lot of time to converge, if it is too large, the algorithm may overshoot the minimum and avoid convergence.

In the previous closed form solution (**subsection 1.1.1**), the hypothesis function was linear in the parameters $\boldsymbol{Y} = \boldsymbol{\Phi}(\boldsymbol{X}) \cdot \boldsymbol{\Theta}$, so we can call this prediction $\hat{\boldsymbol{y}} = \boldsymbol{h}_{\boldsymbol{\Theta}}(\boldsymbol{x})$.

The cost function to be minimized is usually defined as the mean squared error between the prediction and the real data:

$$\text{MSE}(\boldsymbol{X}, h_{\boldsymbol{\Theta}}) = \frac{1}{m} \sum_{i=1}^{m} (\hat{\boldsymbol{y}}_i - \boldsymbol{y}_i)^2 \tag{1.12}$$

The gradient of the cost function, used by all gradient descent algorithms, is defined as:

$$\nabla_{\boldsymbol{\Theta}} \text{MSE}(\boldsymbol{X}, h_{\boldsymbol{\Theta}}) = \begin{bmatrix} \frac{\partial}{\partial \theta_1} \text{MSE}(\boldsymbol{X}, h_{\boldsymbol{\Theta}}) \\ \frac{\partial}{\partial \theta_2} \text{MSE}(\boldsymbol{X}, h_{\boldsymbol{\Theta}}) \\ \vdots \\ \frac{\partial}{\partial \theta_{n_f \times n_y}} \text{MSE}(\boldsymbol{X}, h_{\boldsymbol{\Theta}}) \end{bmatrix} \tag{1.13}$$

The algorithm then updates the parameters at each iteration as:

$$\boldsymbol{\Theta}^{(i+1)} = \boldsymbol{\Theta}^{(i)} - \eta \nabla_{\boldsymbol{\Theta}} \text{MSE}(\boldsymbol{X}, h_{\boldsymbol{\Theta}}) \tag{1.14}$$

1.1.3 Stochastic Gradient Descent

The *Stochastic Gradient Descent* (SGD) is a variant of the GD algorithm that computes the gradient only on one instance at each iteration, instead of on the whole dataset. This makes the algorithm much faster, but the cost function will be much more noisy, and theta will not reach a steady value but instead will oscillate around the minimum. This has the advantage of being more robust to local minimum entrapment, but the disadvantage of never reaching the minimum. To overcome this, the learning rate η can be reduced at each iteration, but this will slow down the convergence.

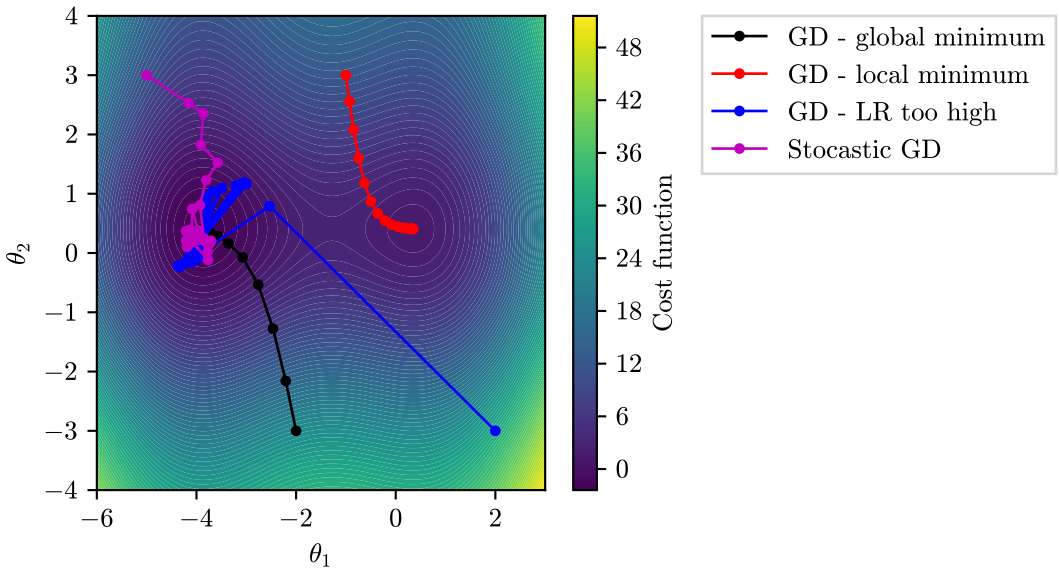


Figure 1.2: *Gradient Descent comparison*

In the **figure 1.2** it is visualized graphically what has been said about Gradient Descent.

1.1.4 Avoid overfitting

The GD algorithm is very powerful, but it can overfit the data. To avoid that, the problem of when to stop the iterations has to be addressed. A common way to do that is to split the dataset into a training set and a validation set. The training set is used to train the algorithm, and the validation set is used to evaluate the performance of the algorithm on new data. The training is stopped when the performance on the validation set starts to degrade, even if the performance on the training set is still improving. This is called *early stopping*. In the **figure 1.3** it is shown an example of early stopping using as metric the *Root Mean Square Error* (RMSE), that is just the square root of MSE, on the validation set.

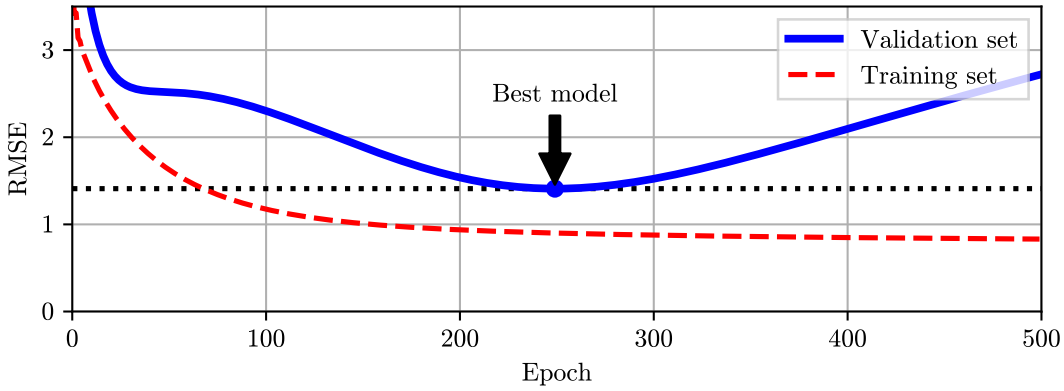


Figure 1.3: *Overfitting example [1, p. 162]*

1.2 Classification

Another common task in ML is classification. In this case, the algorithm is trained to assign a label to a new instance, based on the training dataset of labelled instances. Naively, it aims to define a set of rules that divide the space of the input features in regions, each one associated with a label. The two main approaches are *hard* and *soft* classification. In the former, the algorithm is trained to assign a single label to each instance, while in the latter, the algorithm is trained to output a probability for each label, and the label with the highest probability is assigned to the instance.

Classification is a *supervised* learning task because the training dataset is labelled. The labels can be provided by a human or can be generated by another algorithm. Some classification algorithms are available also in the unsupervised version, where the labels are not provided, and the task is usually novelty detection.

1.2.1 Support Vector Machines SVM

Support Vector Machines are simple but powerful classification algorithms that can be used both for hard and soft classification, with medium size datasets. They are based on the idea of finding the hyperplane that best divides the space of the input features into two regions, each one associated with a label.

The main drawback is that, natively, they can only be used for binary classification (two classes), but there are some extensions that allow to use of them for multiclass classification. Furthermore, as will be explained in **??**, they can be used also for novelty detection (one class). Another limitation is that, being a linear classifier, they can only be used for linearly separable data, but using the *kernel trick*, they can be used also for nonlinearly separable data.

Linear SVM

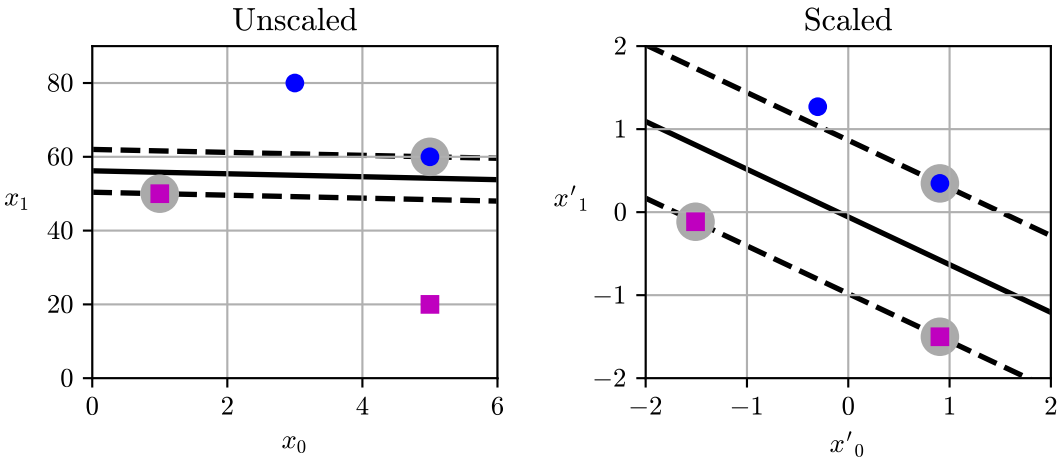


Figure 1.4: *Linear SVM example [1, p. 176]*

Looking at **figure 1.4**, it is possible to visualize what the algorithm does: it finds the plane that separates one class from the other, and vice-versa for the second class. In other words, it finds the most distant parallel hyperplanes that separate the two classes. As evident from the figure, the distance between the hyperplanes (called *margin*) is sensitive to the features scaling. The term "support" derives from the fact that only the instances that are on the margin, define (support) the two planes. Those instances are called *support vectors*, and in the figure are highlighted with a grey circle.

Noninear SVM

As said before, the SVM algorithm can be used also for nonlinearly separable data, using the *kernel trick*. The idea is to project the data into a higher dimensional space, where they are linearly separable, and then use the linear SVM algorithm. The projection is done using a *kernel mapping*.

Let's have a look at what is the function for classifying an instance $\boldsymbol{x}^{(i)}$:

$$t^{(i)} = \begin{cases} -1 & \text{if } \boldsymbol{w}^T\boldsymbol{x}^{(i)} + b < 0 \\ 1 & \text{if } \boldsymbol{w}^T\boldsymbol{x}^{(i)} + b \geq 0 \end{cases} \tag{1.15}$$

The model is trained to find the parameters $\boldsymbol{w}$ and b that:

$$\underset{\boldsymbol{w},b}{\text{minimize}}\ \frac{1}{2}\boldsymbol{w}^T\boldsymbol{w} \tag{1.16}$$

$$\text{subject to } t^{(i)}(\boldsymbol{w}^T\boldsymbol{x}^{(i)} + b) \geq 1 \quad \forall i = 1, \ldots, m \tag{1.17}$$

Since the objective function is convex, and the inequality constraints are differentiable and convex, the solution is the same as the solution of the dual problem [1, p. 188]:

$$\underset{\boldsymbol{\alpha}}{\text{minimize}}\ \frac{1}{2}\sum_{i=1}^{m}\sum_{j=1}^{m}\alpha^{(i)}\alpha^{(j)}t^{(i)}t^{(j)}\boldsymbol{x}^{(i)T}\boldsymbol{x}^{(j)} - \sum_{i=1}^{m}\alpha^{(i)} \tag{1.18}$$

$$\text{subject to } \alpha^{(i)} \geq 0 \quad \forall i = 1, \ldots, m \quad \text{and} \quad \sum_{i=1}^{m}\alpha^{(i)}t^{(i)} = 0 \tag{1.19}$$

Kernel Trick Suppose needing to use a second-degree polynomial mapping, the mapping function is defined as:

$$\phi(\boldsymbol{x}) = \phi(\begin{bmatrix} x_1 \\ x_2 \end{bmatrix}) = \begin{bmatrix} x_1^2 \\ \sqrt{2}x_1x_2 \\ x_2^2 \end{bmatrix} \tag{1.20}$$

Transforming two vectors $\boldsymbol{a}$ and $\boldsymbol{b}$ with the mapping function, to be inserted in **equation 1.18**:

$$\phi(\boldsymbol{a})^T\phi(\boldsymbol{b}) = \begin{bmatrix} a_1^2 \\ \sqrt{2}a_1a_2 \\ a_2^2 \end{bmatrix}^T \begin{bmatrix} b_1^2 \\ \sqrt{2}b_1b_2 \\ b_2^2 \end{bmatrix} = a_1^2b_1^2 + 2a_1b_1a_2b_2 + a_2^2b_2^2 = (\boldsymbol{a}^T\boldsymbol{b})^2 \tag{1.21}$$

So, transforming with a polynomial mapping of degree d, does not require computing the mapping function, but just computing the dot product of the two

vectors and elevating it to the degree d, in the dual problem. There also are other kinds of kernels, resumed in the following:

$$\begin{aligned}
\text{Linear: } & K(\boldsymbol{a}, \boldsymbol{b}) = \boldsymbol{a}^T \boldsymbol{b} \\
\text{Polynomial: } & K(\boldsymbol{a}, \boldsymbol{b}) = (\gamma \boldsymbol{a}^T \boldsymbol{b} + r)^d \\
\text{Gaussian RBF: } & K(\boldsymbol{a}, \boldsymbol{b}) = \exp(-\gamma \|\boldsymbol{a} - \boldsymbol{b}\|^2) \\
\text{Sigmoid: } & K(\boldsymbol{a}, \boldsymbol{b}) = \tanh(\gamma \boldsymbol{a}^T \boldsymbol{b} + r)
\end{aligned}$$

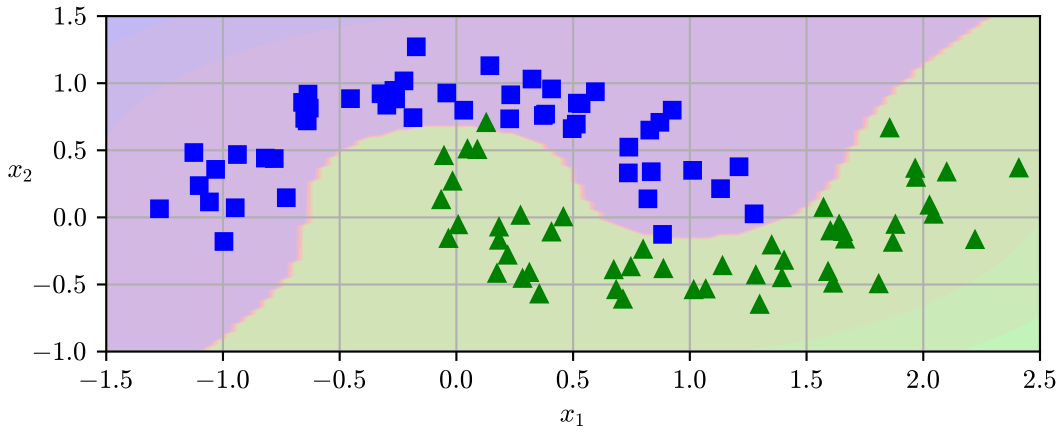


Figure 1.5: *Kernel Trick example [1, p. 180]*

The **figure 1.5** shows an example of SVM classification of data that are not linearly separable.

This topic seems unrelated to the scope of this thesis, but in **??** we will see how to use the SVM algorithm for novelty detection, as a one-class classifier.

1.2.2 Decision Trees DT

Decision Trees are very powerful classification algorithms that can also be used for regression, thinking of feature values as classes. The classification process is based on a tree structure, where each sample starts from the root node, and is filtered through a bunch of *if - then* statements until it reaches a leaf node, that outputs the predicted class. In **figure 1.6**, it is illustrated the structure of a very simple binary tree with only a split node and three leaf nodes.

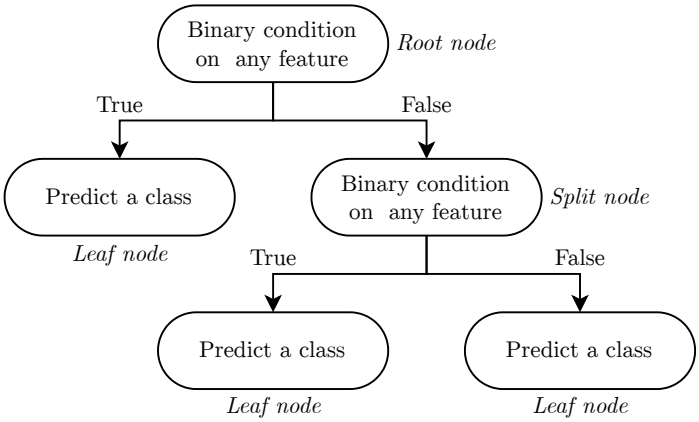


Figure 1.6: *Decision Tree structure*

Gini impurity The classification algorithm is hence very simple, the ML part is the training process. Let's consider a leaf node, and imagine processing all the training samples through the tree. Ideally, all the samples that reach the leaf node (and any other leaf node) should have the same class. This is possible, but a tree that does that is most likely very overfitted to the training dataset and will not perform well on future data. Anyway, the aim of training is to obtain a tree close enough to the ideal one, without overfitting. To do that, there exists a metric called *Gini impurity* that assumes a value of zero if the leaf node is pure (all the samples that reach it have the same class), or a positive value $\in$ (0,0.5] that measures how different the classes in the node are, 0.5 being the maximum value that means that all the classes are present in the node with equal frequency. The mathematical definition is the following:

$$G_i = 1 - \sum_{k=1}^{n} p_{i,k}^2 \tag{1.22}$$

where $p_{i,k}$ is the ratio of class k instances among the training instances in the i^{th} node.

Then the training procedure tries to grow a tree defining the binary conditions that minimizes the weighted average of the Gini impurity of the two child nodes, so the cost function is:

$$J(k, t_k) = \frac{m_{\text{left}}}{m} G_{\text{left}} + \frac{m_{\text{right}}}{m} G_{\text{right}} \tag{1.23}$$

where k is the feature index, t_k is the threshold value, m_{left} and m_{right} are the number of instances in the left and right child nodes, and G_{left} and G_{right} are the Gini impurity of the left and right child nodes.

A common way for minimization of the cost function is to use the *Classification and Regression Tree* (CART) algorithm, which is a greedy algorithm that searches

for the optimal split at each node, but not for the global optimal tree. The algorithm complexity is $\mathcal{O}(n \times m \log_2(m))$.

Entropy Another metric that can be used instead of Gini impurity inside the same cost function is the *entropy* of the node, which is defined as:

$$H_i = -\sum_{k=1}^{n} p_{i,k} \log_2(p_{i,k}) \tag{1.24}$$

This renders trees very similar to the ones obtained using Gini impurity, but the entropy is slightly slower to compute, due to the logarithm. however, it tends to produce slightly more balanced trees [2].

Avoid overfitting To avoid overfitting the data, the CART algorithm implementation in `sklearn` has some hyperparameters that can be tuned:

- `max_depth`: the maximum depth of the tree;
- `min_samples_split`: the minimum number of samples a node must have before it can be split;
- `min_samples_leaf`: the minimum number of samples a leaf node must have;
- `max_leaf_nodes`: the maximum number of leaf nodes;
- `max_features`: the maximum number of features that are evaluated for splitting at each node.

Increasing the `min` bound, or decreasing the `max` bound, will regularize the model, and reduce the risk of overfitting. In **figure 1.7** it is shown an example of overfitting, where the left plot shows the decision boundaries of a tree with no regularization, and the right plot shows the decision boundaries of a tree with regularization.

Regression As anticipated, the DTs can also be used for regression, in this case, the cost function is the MSE of the predicted value in the leaf node:

$$J(k, t_k) = \frac{m_{\text{left}}}{m}\text{MSE}_{\text{left}} + \frac{m_{\text{right}}}{m}\text{MSE}_{\text{right}} \tag{1.25}$$

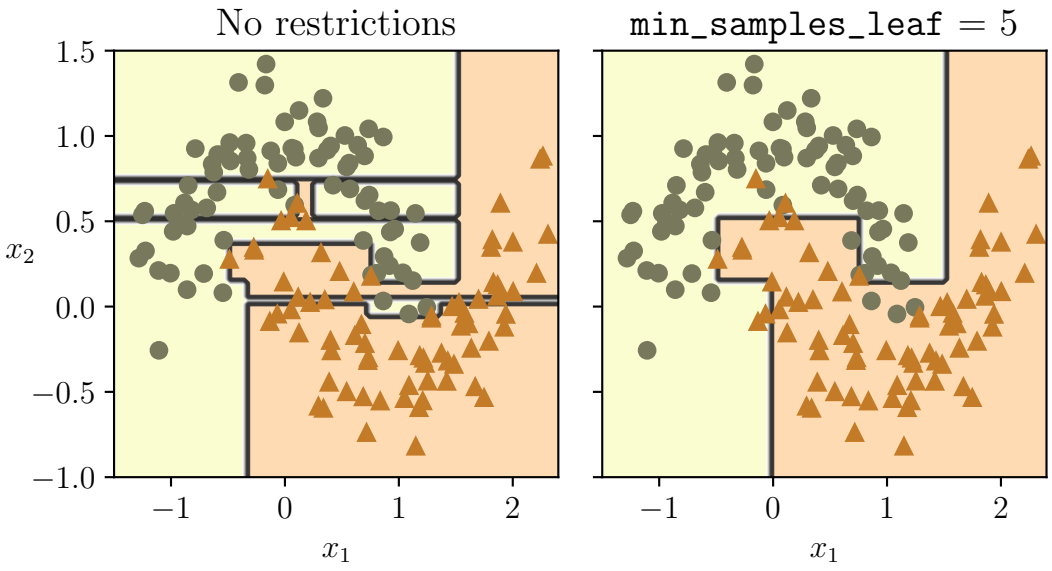


Figure 1.7: *Decision Tree overfitting example [1, p. 203]*

Advantages and limitations The main disadvantages of DTs are that the classification procedure uses thresholds on the value of the features (cutting the hyperspace in orthogonal hyperplanes), so they are sensitive to axis orientation, and they are very sensitive to small variations in the training data. They are also very sensitive to the hyperparameters, so a small variation in constraints leads to very different trees. The main advantages are that they are very fast to train, and the resulting model is very fast to make predictions, they are very easy to understand and visualize and they do not require any feature scaling or centring.

1.2.3 Random Forests RF

The high sensitivity of the DTs to small variations in the training data, can be reduced using the *Random Forests* (RF) algorithm. The idea is to train a bunch of DTs on different random subsets of the training data, and then to average their predictions. The subsets of the training set are usually picked randomly with replacement, this technique is called *bagging* (short for *bootstrap aggregating*).

The benefits of using more threes on subsets of the training data are shown in the **figure 1.8**. The left plot shows the decision boundaries of a single DT, and the right plot shows the decision boundaries of a RF with 500 trees.

Again, this topic seems unrelated to the scope of this thesis, but in **??** we will see how to use the RF algorithm for novelty detection, exploiting the fact that

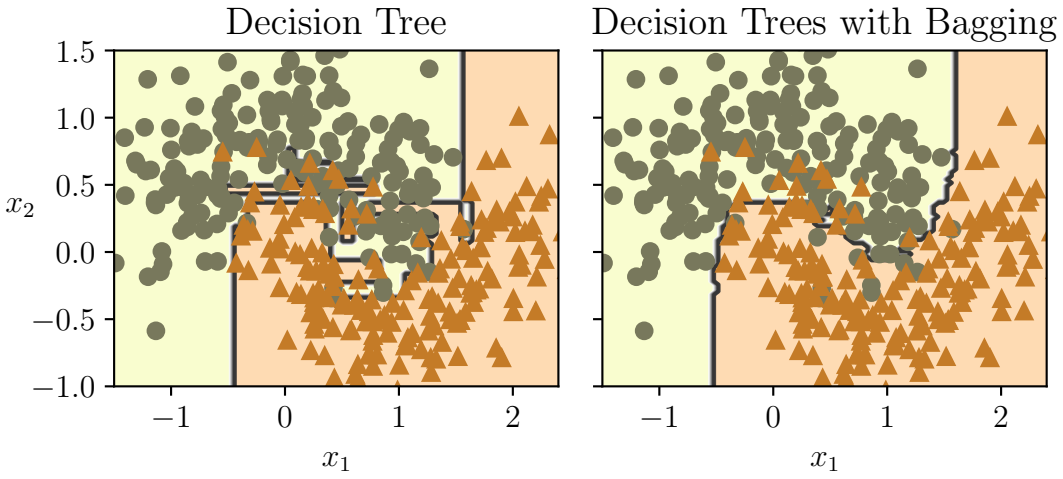


Figure 1.8: *Random Forest example [1, p. 218]*

outliers are usually more isolated (require more split nodes to be reached) than the normal instances.

Bibliography

[1] Aurelien Geron. *Hands-On Machine Learning With Scikit-Learn, Keras & TensorFlow*. English. O'Reilly Media, 2022, p. 850. ISBN: 9781098125974 (cit. on pp. 1, 7–10, 13, 14).

[2] Sebastian Raschka. «Why are implementations of decision tree algorithms usually binary and what are the advantages of the different impurity metrics?» In: *Machine Learning FAQ* (2013). Available at: `http://sebastianraschka.com/faq/docs/decision-tree-binary.html` (cit. on p. 12).