Machine Learning Modeling

Ntigkaris Alexandros

antigkar@auth.gr

Contents

1 Linear Models

1.1 Linear Regression

The first method proposed is none other than linear regression. Linear regression is most often called simple linear regression when the predictors' number is one and multiple linear regression otherwise. Linear regression is most commonly used as a baseline model in order to get a first feel around how off our measurements are from the ground-truth.

Linear regression belongs to the family of linear predictor function methods. Linear predictor functions are linear functions of a finite set of coefficients and independent variables, whose value derives the estimation on a dependent variable y:

$$f(i) = \beta_0 + \beta_1 x_{i1} + \cdots + \beta_m x_{im} \tag{1}$$

$$\iff f(i) = \boldsymbol{x_i^T} \cdot \boldsymbol{\beta} \tag{2}$$

Linear regression tries to find a linear relationship between a target variable and a set of independent variables. Contrary to linear predictor functions, the linear relationship between y and x cannot be fully described without the introduction of an error or noise term ϵ. The errors are required to be randomly distributed by nature, if the relationship between the two variable kinds is linear. Otherwise, the use of linear regression is not deemed appropriate. Linear regression can be represented mathematically as:

$$\boldsymbol{y} = \boldsymbol{X} \cdot \boldsymbol{\beta} + \boldsymbol{\epsilon} \tag{3}$$

where:

- the target vector $\boldsymbol{y} = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix}$
- the independent variable matrix $\boldsymbol{X} = \begin{pmatrix} 1 & x_{11} & \cdots & x_{1m} \\ 1 & x_{21} & \cdots & x_{2m} \\ \vdots & \vdots & \vdots & \vdots \\ 1 & x_{n1} & \cdots & x_{nm} \end{pmatrix}$
- the coefficient vector $\boldsymbol{\beta} = \begin{pmatrix} \beta_0 \\ \beta_1 \\ \vdots \\ \beta_m \end{pmatrix}$

- the error vector $\boldsymbol{\epsilon} = \begin{pmatrix} \epsilon_1 \\ \epsilon_2 \\ \vdots \\ \epsilon_n \end{pmatrix}$

The best linear fit can be attained when we find the optimal weights $\boldsymbol{\beta}$ that correspond to the minimum error $\boldsymbol{\epsilon}$. We remind here that error can be written as:

$$\boldsymbol{\epsilon} = \boldsymbol{y} - \hat{\boldsymbol{y}} \tag{4}$$

$$\Rightarrow \ \boldsymbol{\epsilon} = \boldsymbol{y} - \boldsymbol{X} \cdot \boldsymbol{\beta} \tag{5}$$

with the expected value of the errors $\boldsymbol{\epsilon_i}$ being zero and their variance equal to $\sigma^2_{\boldsymbol{\epsilon}}$. Therefore in order to acquire the minimum error, we use the least squares method, which states that the total sum S of the squares of the errors must be minimum:

$$S = \sum_{i=1}^{n} \epsilon_i^2 \tag{6}$$

$$\Rightarrow \ \hat{\boldsymbol{\beta}} = argmin\ S(\boldsymbol{\beta}) \tag{7}$$

Hence,

$$\frac{\partial S}{\partial \beta_j} = 0, \quad j = 1, \ldots, m \tag{8}$$

$$\Rightarrow \ 2\sum_{i=1}^{n} \epsilon_i \frac{\partial \epsilon_i}{\partial \beta_j} = 0 \tag{9}$$

In the simplest case of $\hat{\boldsymbol{y}} = \boldsymbol{\beta_0}$, then the result of the least squares would be none other than the arithmetic mean of the data. In the case of an 1D feature vector, linear regression can be expressed as:

$$y = \beta_0 + \beta_1 X \tag{10}$$

Best values for weights β_0, β_1 can easily be obtained from:

$$\hat{\beta}_1 = \frac{\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})}{\sum_{i=1}^{n}(x_i - \bar{x})^2} \tag{11}$$

$$\hat{\beta}_0 = \frac{\sum_{i=1}^{n} y_i - \hat{\beta}_1 \sum_{i=1}^{n} x_i}{n} \tag{12}$$

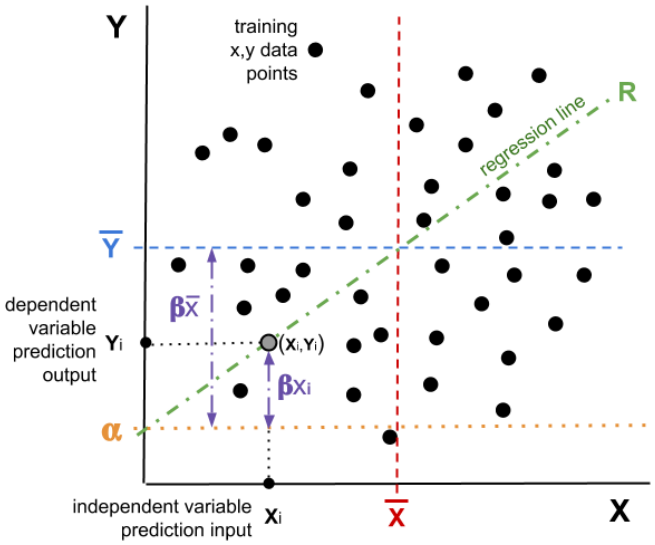

Figure 1: 1D feature vector linear regression.

1.2 Perceptron

Perceptron, also known as McCulloch-Pitts neuron, is another member of the linear predictor function methods. Originally conceived in 1943 as an imitation to the biological neuron, perceptron was used as a linear classifier for images. The machine itself used 20×20 cadmium sulfide photocells to construct 400 px images. These photocells were initially connected randomly with potentiometers and weight updates were performed using electric motors. The perceptron algorithm can easily be expressed as:

$$\boldsymbol{y} = \sum_{i=0}^{n} \boldsymbol{x_i w_i} \tag{13}$$

$$\Rightarrow \boldsymbol{y} = \boldsymbol{x} \cdot \boldsymbol{w} \tag{14}$$

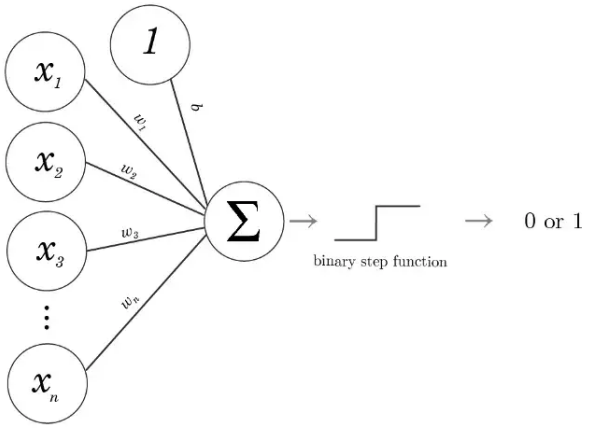

Figure 2: The original perceptron configuration.

The modern perceptrons or the simplest form of a neural network, use stochastic gradient descent as the weight-update algorithm, also known as back-propagation:

$$\boldsymbol{w} := \boldsymbol{w} - \eta \nabla S(\boldsymbol{w}) \tag{15}$$

where:

- S is the optimization objective function, as seen in Eq. 6,
- η is the gradient stepsize, also called learning rate.

The meaning behind the term "stochastic" revolves around the sacrifice being made between computing the true gradient of the data and computing the sample's batch gradient, as in our case. Depending on the learning rate choice, stochastic gradient descent will mostly converge to a global minimum when the objective function is convex or pseudoconvex. The main difference between linear regression and stochastic gradient descent is that while linear regression computes the exact analytical solution but performs slower for large datasets, stochastic gradient descent is faster but displays fluctuations during training and can fail to converge properly without careful parametric tuning.

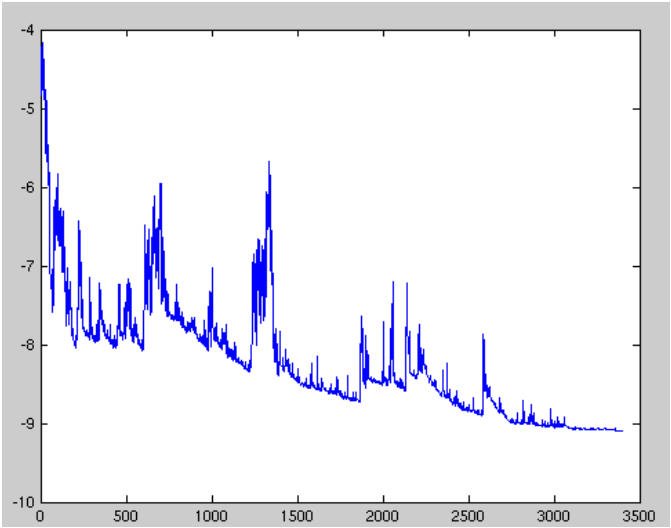

Figure 3: Perceptron's back-propagation training displays fluctuations due to the stochastic nature of the algorithm.

2 Deep Neural Networks

2.1 Feedforward Networks

Despite the success of linear models in a plethora of problems, not all data can be treated using linear approaches and some data can only be described with more complex non-linear models. Thus, in order to introduce the power of non-linearity, feedforward networks were used. The term "feedforward" means that information input move in a forward direction through the node layers, resulting in an output. Each hidden layer between the input and the output nodes, applies an activation function to the information flow. The purpose of the activation function is to transform the information into a non-linear manner, with the hope that the network will unravel the data' complex nature in future iterations.

The most common activation function is the sigmoid:

$$f(x) = \frac{1}{1 + e^{-x}} \tag{16}$$

$$f'(x) = f(x)(1 - f(x)) \tag{17}$$

The reason this function is widely being used is that it transforms the input in a probabilistic distribution $[0, 1]$, which is a really helpful in binary classification tasks, while it provides the gradient calculation step with a very easy-to-compute derivative.

Other interesting activation function choices include the rectified linear unit function (ReLU):

$$f(x) = max(0, f(x)) \tag{18}$$

$$f'(x) = \begin{cases} 1 & \text{if } x > 0 \\ \\ 0 & \text{else} \end{cases} \tag{19}$$

This function acts as a stricter and more efficient form of the sigmoid. ReLU will always return a positive x value, which allows for more constant non-vanishing gradients, while sigmoid's exponent tends to saturates at either 0 or 1 as x gets smaller or larger respectively. Lastly, sigmoid outputs result in dense non-negative garbage filled representations, while ReLU gives sparse and clearer matrix representations instead. The above reasons make the network's computations simpler, faster and more efficient.

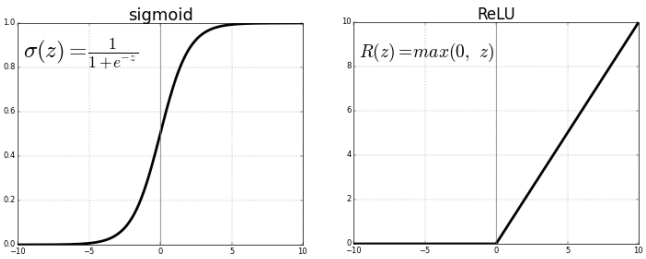

Figure 4: Activation functions.

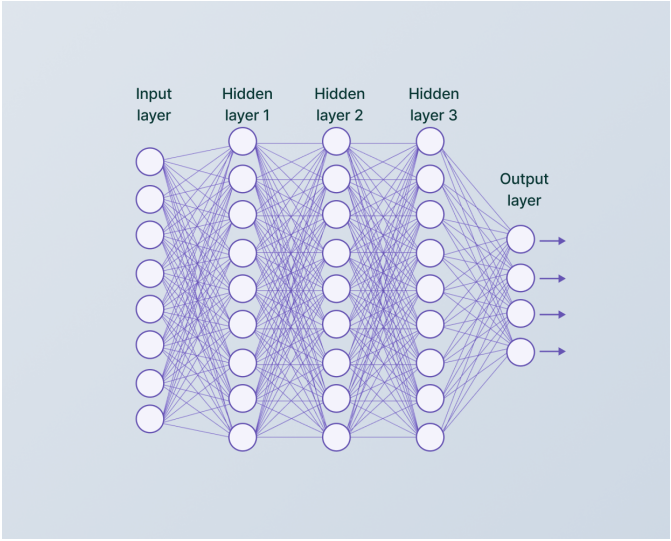

Figure 5: Deep neural network typical architecture.

2.2 Recurrent Networks

MLPs' non-linearity is a great start to achieve results better than the baseline but when focusing on time-series and generally sequential data, MLPs fail to extract the correlation between past and future values. In order to solve that, we use recurrent neural networks (RNN). Contrary to MLP taking batches of values per iteration, RNN takes batches of sequences of certain length. Therefore, during each iteration the RNN input will be a sequence of N values or batches of sequences with shape $b \times 1 \times N$. RNN begins with an uninitialized information representation of fixed size, called the hidden state. From there on, every element of the sequence will be fed iteratively in the RNN updating the hidden state with each pass. At the end of the iterations, the final hidden state will contain all the information context, which was fed sequentially to the model.

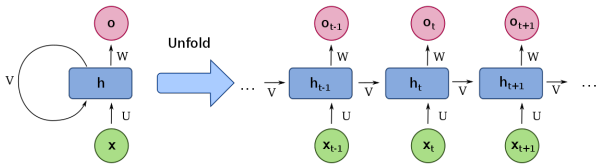

Figure 6: Recurrent neural network architecture.

Mathematically, the fully recurrent neural network can be presented as:

$$a_h^t = \sum_{i=1}^{I} w_{ih} x_i^t + \sum_{h'=1}^{H} w_{h'h} b_{h'}^{t-1} \tag{20}$$

$$b_h^t = f_h(a_h^t) \tag{21}$$

$$a_k^t = \sum_{h=1}^{H} w_{hk} b_h^t \tag{22}$$

where:

- a_h^t the output of the hidden state h at time-step t (lower vertical motion)
- b_h^t the activation function's output of a_h^t (horizontal motion)
- a_k^t the model's $k^t h$ output at time-step t (upper vertical motion)

However, again due to the nature of neural networks, the influence of a given input's information can either diminish or explode exponentially as we cycle through the RNN loop. Even though ReLU is the solution behind this problem in MLPs, here due to the recurrence, the problem exacerbates. The next level of RNNs goes by the name LSTM which stands for Long Short-Term Memory network. As with RNN's ability to store short-term information but decay their importance in the long run, LSTM "lengthens" the short-term capacity of a typical RNN. This is done via the addition of an extra hidden state called the cell state, which is responsible for what goes into the hidden state output. Changes in the cell state are performed via four gates called the input, forget, cell, and output gates. These gates regulate how information are distributed in the cell state.

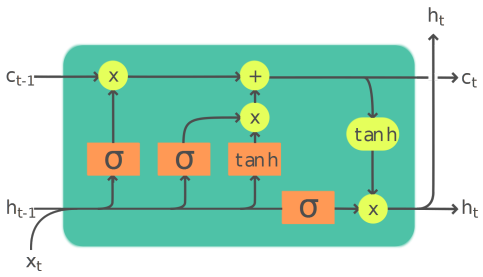

Figure 7: LSTM network architecture.

When the input gate is activated, the hidden state acquires the input's information context. As long as the forget gate remains open, the input's information are preserved in the hidden state processing. The output gate can activate to deliver the state as output. Once the forget gate is closed, new input information are written in the hidden state.

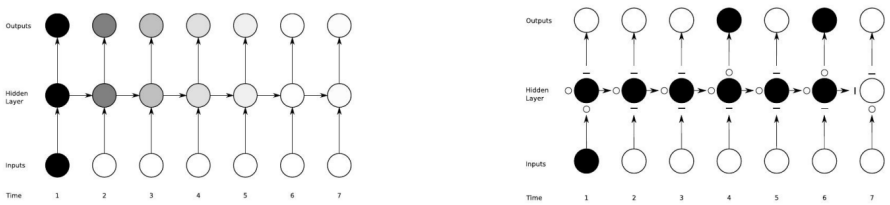

Figure 8: Comparison between RNN's and LSTM's ability to recall information context from earlier inputs as time-steps increase.

Each sequence element input inside the model, passes through the following functions corresponding to the four information gates:

$$i_t = \sigma(W_{ii}x_t + b_{ii} + W_{hi}h_{t-1} + b_{hi}) \tag{23}$$

$$f_t = \sigma(W_{if}x_t + b_{if} + W_{hf}h_{t-1} + b_{hf}) \tag{24}$$

$$g_t = tanh(W_{ig}x_t + b_{ig} + W_{hg}h_{t-1} + b_{hg}) \tag{25}$$

$$o_t = \sigma(W_{io}x_t + b_{io} + W_{ho}h_{t-1} + b_{ho}) \tag{26}$$

The updates to the two states are performed using the outputs of the aforementioned function calls:

$$c_t = f_t \odot c_{t-1} + i_t \odot g_t \tag{27}$$

$$h_t = o_t \odot tanh(c_t) \tag{28}$$

where $\odot$ is the Hadamard product (element-wise multiplication).

From Eq. 23, 24, 25, 26 we can observe how both input x and previous hidden state h information concatenate and pass through the input, forget, cell, and output gates. Then in Eq. 27 we see that for the cell state, the forget gate influences how much of the previous cell information must be kept, while the cell gate regulates how much of the new input information should be added. Lastly in Eq. 28, the hidden state is updated according to the cell state's contents.

2.3 Convolutional Networks

Convolutional networks (CNN) are another type of deep neural network. Their main utility is capturing spatially locally encoded information. To illustrate this better, we first have to bear in mind that CNNs typically work as image recognition models. Explicitly, images are nothing more than 2D matrices of pixels encoded as numerical values. However, human beings do not classify images solely based on color depth and distribution, but on characteristics about the individual objects depicted on them. Those characteristics are information encoded spatially in the image itself. For example, in a dataset depicting cats and dogs, the animal's height and silhouette is a characteristic that is locally encoded in the majority of the image's surface. On the contrary, the animal's ears or facial features are locally encoded in a much more smaller scale. Therefore, different amount of attention needs to be paid to different spatially distributed parts of the image.

A convolutional network operates like an MLP, but here the hidden layers apply the convolution operation to the inputs. In order for the convolution to take place, a sliding kernel matrix must be defined. The kernel will slide with a step called stride through the input 2D matrix and each time it will perform an element-wise multiplication with the kernel's projection on the matrix space. Finally it will output the convolved 2D matrix. In order to achieve a convolved matrix of specific size, padding can be added to the input matrix. The size of the kernel matrix is analogous to the size of the feature or characteristic we are trying to extract from the image.

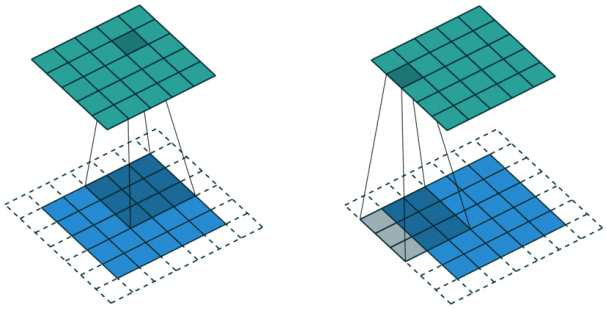

Figure 9: The kernel slides through the matrix performing convolution operations.

In a similar manner, convolution can be performed on 1D vectors as well. Given an input x of length n and a kernel h of length k, the convolution between x and h for stride s can be expressed as:

$$y(n) = \begin{cases} \sum_{i=0}^{k} x(n-i)h(i) & \text{if } n = k-1 \\ \sum_{i=0}^{k} x(n-i+s-1)h(i) & \text{else} \end{cases} \tag{29}$$

For example, for the following unpadded vector x and kernel h with stride $s = 1$, the resulting vector y will be:

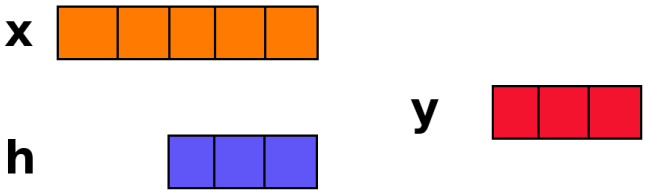

Figure 10: Convolution example.

$$\begin{aligned} y(2) &= x(2)h(0) + x(1)h(1) + x(0)h(0) \\ y(3) &= x(3)h(0) + x(2)h(1) + x(1)h(0) \\ y(4) &= x(4)h(0) + x(3)h(1) + x(2)h(0) \end{aligned}$$

3 Ensembles

3.1 Decision Trees

Decision trees can be thought as optimization search structures in classification or regression tasks. In the same way that tree data structures operate, we begin with a root node containing the whole training dataset. Then for every feature in the dataset sample examined, we perform a search for the best possible split in the dataset. The best split is defined when the split dataset sample's rule metric becomes minimum. When a best split is found, the node splits into a pair of left and right nodes. The splitting keeps taking place until stopping criteria (e.g. max tree depth) are met.

For classification purposes the rule metric can be either be information gain or gini index. For regression tasks, the chosen metric is variance reduction. The variance of a sample of continuous variables is defined as:

$$Var = \frac{\sum_{i=1}^{N}(x_i - \mu)^2}{N} \tag{30}$$

where $\mu = \frac{1}{N}\sum_{i=1}^{N} x_i$.

In practice, variance reduction of a candidate split of a parent node into two child nodes, is expressed as:

$$VarRed = Var_P - \left(\frac{n_C^L}{n_P} Var_C^L + \frac{n_C^R}{n_P} Var_C^R \right) \tag{31}$$

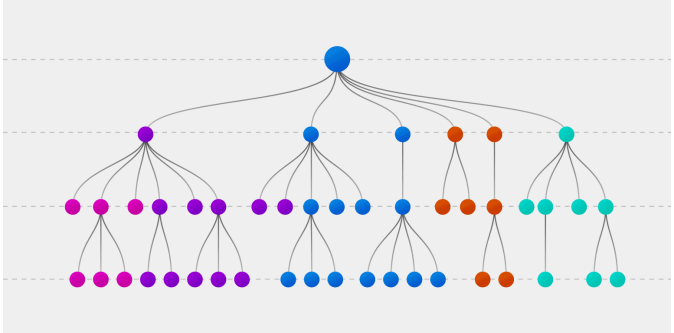

Figure 11: Decision tree structure.

3.2 Random Forests

Random forests are an extension to the decision trees logic, where an ensemble of trees is used. The algorithm applies bagging. Given a dataset $[y, X]$, bagging assigns B times, random

samples $[y_b, X_b]$ with replacement (bootstrap) to fit each tree. At the end of training, the unseen samples $[y_u, X_u]$ are used for inference and the random forests' output will be the averaged predictions. Mathematically, the output of a random forest is defined as:

$$\hat{f} = \frac{1}{B} \sum_{b=1}^{B} f'_b(x_u) \tag{32}$$

The reason behind using bootstrap is that it de-correlates the trees and subsequently reduces the variance in the final predictions. In order to make this de-correlation even stronger, random forests also apply feature bagging. For every training tree, a random subset of features with replacement is selected, instead of the whole feature list. Therefore, by training different trees in different features and with different data, a further reduction in variance is achieved.

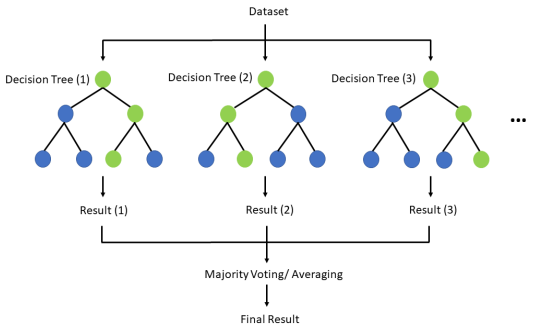

Figure 12: Random forest algorithm.

3.3 Gradient Boosting

Gradient boosting algorithms are tree algorithms applying iterative functional gradient descent, meaning that they are trying to optimize a cost function by iteratively choosing a model function f that points in a more optimal negative gradient direction. Contrary to random forests which work parallelly by ensembling several learners into an averaging model, gradient boosting machines work sequentially, starting with a single weak learner and iteratively boosting them into a stronger learner by correcting the errors of the previous one.

Mathematically, we can formulate this method as following: The goal of the problem is to teach an estimator F to predict values $\hat{y}$ by minimizing the error $\frac{1}{n}\sum_i(\hat{y_i} - y_i)^2$, where n is the number of samples. Let's suppose a gradient boosting algorithm with M iterations. For m^{th} iteration, $F_m(x_i)$ will return $\hat{y_i}$. In order for $\hat{y_i} = y_i$ and the model to perform best, we should add another estimator (weak learner) $h_m(x_i)$ so that:

$$F_{m+1}(x_i) = F_m(x_i) + h_m(x_i) = y_i \tag{33}$$

$$\Rightarrow h_m(x_i) = y_i - F_m(x_i) \tag{34}$$

Therefore:

$$L = \frac{1}{n}\sum_{i=1}^{n}(y_i - F(x_i))^2 \tag{35}$$

$$\Rightarrow -\frac{\partial L}{\partial F(x_i)} = \frac{2}{n}h_m(x_i) \tag{36}$$

Hence, the algorithm each time, will sequentially add a new learner $h_{m+j}(x_i)$ to correct the loss derived from the previous learner.

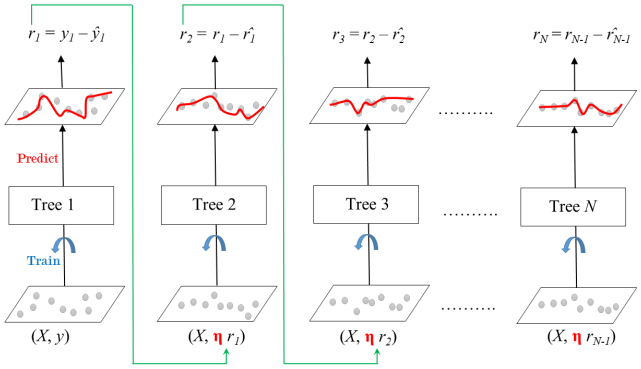

Figure 13: Gradient boosting algorithm.

4 Metrics

A key aspect in modeling is the assessment of models' outputs and how well they correspond to reference values. In literature and statistics, the most well-known metrics are the mean absolute error (MAE), root mean squared error (RMSE) and coefficient of determination (R^2). Each of these metrics holds its own significance in decision-making and is influenced by the dataset in a unique way than the rest.

4.1 Mean Absolute Error

MAE is defined as the mean absolute difference between the ground-truth values and the predicted values:

$$MAE = \frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i| \tag{37}$$

where:

- n is the sample size,
- y_i is the ground-truth value,
- $\hat{y}_i$ is the predicted value.

MAE represents the average absolute vertical or horizontal difference between each point in the $(y, \hat{y})$ scatter plot. Thus all errors e_i will be assigned the same weights regarding to the model's decision making process. In simple terms, MAE tells us what's the most expected difference of our model's performance from the ground-truth.

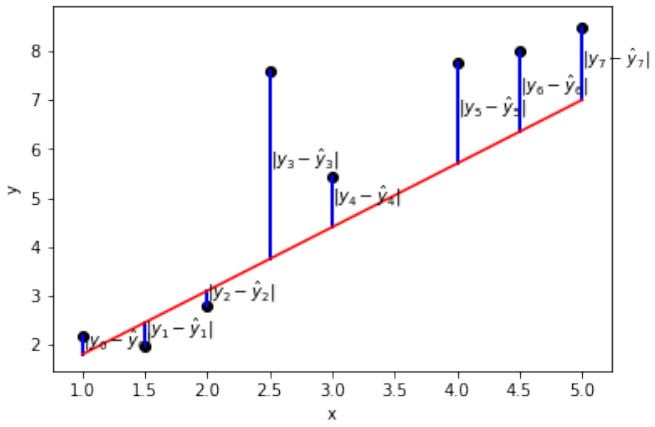

Figure 14: Differences between the predicted and ground-truth values for MAE.

4.2 Root Mean Squared Error

In contrast, RMSE is defined as the root mean absolute difference between the squares of the ground-truth values and the predicted values:

$$RMSE = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2} \tag{38}$$

where:

- n is the sample size,
- y_i is the ground-truth value,
- $\hat{y}_i$ is the predicted value.

Because RMSE squares the errors, it applies a weighted factor on them, thus outliers or values with large differences are more prevalent and noticeable using this metric. Moreover, by its definition RMSE will always be larger than MAE number-wise, but RMSE's vector space is euclidean, contrary to MAE which depicts raw distances. Again to sum things up, RMSE is just like MAE but with increased sensitivity to outliers.

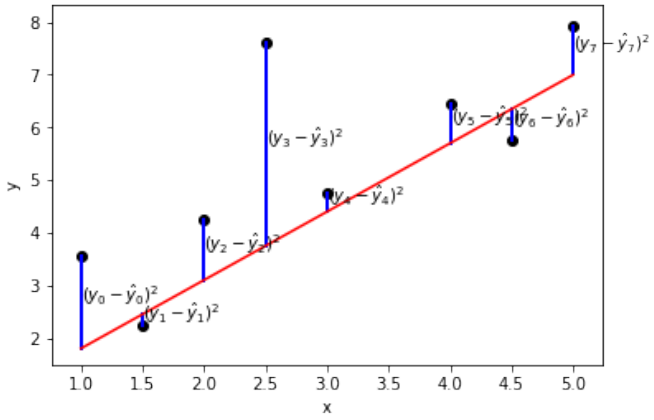

Figure 15: Differences between the predicted and ground-truth values for RMSE. We observe that outliers are significantly noticeable while the rest of the values are flattened.

4.3 Coefficient of Determination

Another metric is the coefficient of determination, also known as R^2 score. R^2 is derived as the square of Pearson's r sample correlation coefficient. Because its value ranges from 0 to 1, it provides an easier way to infer a model's goodness-of-fit than MAE and RMSE. However,

it should be noted that R^2 can sometimes take negative values and that can happen when a non-linear function is used to fit the data. Mathematically, it can be expressed as:

$$R^2 = 1 - \frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{\sum_{i=1}^{n}(y_i - \bar{y})^2} \tag{39}$$

where:

- n is the sample size,
- y_i is the ground-truth value,
- $\hat{y}_i$ is the predicted value,
- $\bar{y}$ is the average ground-truth value.

According to the above definition, the most basic baseline model that is only able to "predict" the data mean, will have a R^2 of 0. A perfect model whose predicted value will always match the ground-truth value will have a R^2 of 1. Anything in-between is measured similarly to the former metrics, but with the addition of a weight factor which measures the difference of the predicted value from the data mean, a distance proportional to the variance. Thus, R^2 can be seen as an indicator about the variability of the ground-truth accounted for, by the model.

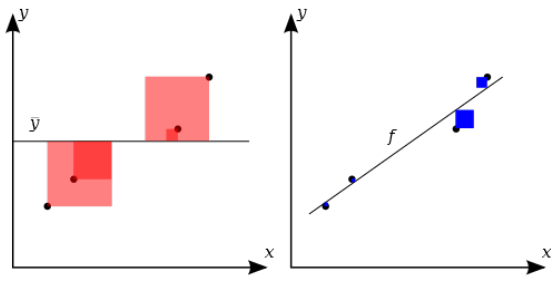

Figure 16: Representation of how the squared differences balance around the mean (left) versus around the regression line (right).

4.4 Index of Agreement

In 1981 Willmott proposed a variation of the R^2 score, called the index of agreement (IoA). In a similar manner, IoA score varies between 0 and 1 but can also detect additive, proportional differences in the groud-truth and predicted means and variances, while being sensitive to extreme values due to the squared terms. Mathematically, it can be formulated as:

$$IoA = 1 - \frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{\sum_{i=1}^{n}(|y_i - \bar{y}| + |\hat{y}_i - \bar{y}|)^2} \tag{40}$$

where:

- n is the sample size,
- y_i is the ground-truth value,
- $\hat{y}_i$ is the predicted value,
- $\bar{y}$ is the average ground-truth value.

4.5 Other Metrics

Finally other metrics considered, are variations of the aforementioned ones, like the mean absolute percentage error (MAPE):

$$MAPE = 100\% \frac{1}{n} \sum_{i=1}^{n} |\frac{y_i - \hat{y}_i}{y_i}| \tag{41}$$

or the unbiased root mean squared distance (uRMSD):

$$uRMSD = \sqrt{\frac{1}{n} \sum_{i=1}^{n} \Big((\hat{y}_i - \bar{\hat{y}}) - (y_i - \bar{y})\Big)^2} \tag{42}$$