1 Decision tree

1.1 Principle

A decision tree is a binary tree in which each node corresponds to a subset of the data in the parent node. The splits used to create the child nodes are found by minimizing a cost function which represents how 'pure' are the nodes, i.e. how mixed are the classes of the data in the node. This cost function is computed on each split of each feature in the training set. The data subset in the node is then distributed between the two children nodes according to the fact that the value at the chosen feature is lower or higher than the value chosen for the split.The objective is to obtain nodes that are as pure as possible, hopefully with only one class in it.

Once the tree is trained, the prediction is simply done by travelling across the tree from node to node according to the split defined in the current node.

1.2 Algorithm

1.2.1 Main idea

The algorithm is pretty simple. For each feature in the dataset, we begin by sorting the values vector of that feature in ascending order. Then, for each possible value in this vector, we split the data in left subset (points having value lower than this) and right subset (points having value higher than this) and compute the score on these subsets. The value giving the subsets with the highest score among all features is selected as split point.

This step is repeted for each node, giving birth to the left and right children nodes, and the process is repeted recursively until all the nodes are pure or a certain depth is reached.

1.2.2 Cost function

The cost function used here is the GINI purity score. This score tells how pure is a node, i.e. how similar the classes in the node are. A node with only one classe in it would then have a GINI score of 1.

The GINI score is computed as follows:

$$G = \sum_{i=1}^{c} (p_i)^2$$

With c the number of classes in the dataset, and p_i the probability for a datapoint of being of class i.

This score holds for each subset given by the split. The compute the total score for the split, the GINI score is computed for each subset, and then weighted by the proportion of the data of the parent node that is present in each subset, and summed. So, if m is the size of the starting dataset, i the size of the left subset and j the size of the right subset obtained by splitting the starting dataset:

$$G_{split} = \frac{i}{m} G_{left} + \frac{j}{m} G_{right}$$

1.2.3 Computation tricks

To make the training process faster, one trick that can be used is to keep track of the classes count at each split tested, in order not to have to recompute everything at each step. So, in every split, instead of scanning the whole array to find values that are lower than the split point and values that are higher than it, we simply check the class label of the current split, add 1 to the count of this class on the left subset, and remove 1 from the count of this class in the right subset. This trick allows to find the new subsets composition in $O(1)$ complexity, where scanning the array would be $O(n)$.

2 Neural network

2.1 Principle

A neural network is a set of layers composed of nodes called neurons, connected to each other by weighted connections. These layers are of three type: one input layer, in which the iput is given, one or more hidden layers, and an output layer, giving the result of the prediction. One more node can be added to each layer: the bias node.

The idea behind neural network is to adapt the weights of the connections between the nodes, as well as the biases, to get an accurate prediction.

2.2 Algorithm

2.2.1 Main idea

To goal of the algorithm is to find a good set of weights and biases to get an accurate prediction. The principal tool used to achieve this goal is called backpropagation. Backpropagation is an algorithm using the error on the output layer to retro-compute the errors generated by the weights and biases in order to adapt them step by step.

The first step in a neural network is the feedforward step, where we will compute the output given the current parameters. To do so, we first initialize the weights at random. Then, each node is computed as the combination of each node of the previous layer, weighted by the given weight connecting this node to the corresponding node of that previous layer. Then, the computed node is 'activated' by a so-called activation function that maintains it in a certain range (for example, between 0 and 1 with a sigmoid activation function). This process is repeted until the output node, where the obtained values gives the output.

The problem then becomes an optimisaton problem: we define a cost function, reflecting the error in the output layer, and we will try to minimize it. This is done by finding an area where the derivative of this function is close to 0, meaning the cost function cannot decrease anymore (at least in this area as this could be a local minimum). This is where backpropagation comes into play. By chain rule, we can compute the error on the layer $L-1$ thanks the to error on the layer L, i.e., we can get the error on each layer given the error on the output. With this process, we will adapt the weights by passing the inputs of the training set through the network one by one.

2.2.2 Activation function

The activation function used here is the classical sigmoid activation. It is given by the following formula:

$$\sigma = \frac{1}{1+e^{x}}$$

This function is classicaly used in neural networks. It keeps value in a range between 0 and 1. This allows to add non-linearity in the network, allowing to perform non-linear mappings.

2.2.3 Cost function

The cost function used here is the cost-entropy function. This cost-function gives better results than a simple MSE score because it avoids slow-down in learing by not having a partial derivative dependant of the activated derivative that can be close to 0 due to the nature of sigmoid activation. The cross-entropy function is computed as follows:

$$C = -\frac{1}{n}\sum_{x}[yln_a + (1-y)ln(1-a)]$$