Decision Trees, Random Forests, Data and Features[*]

March 9, 2022

1 Introduction

In the previous lectures we dealt with the design of linear classifiers described by linear discriminant functions $g(\mathbf{x})$,

$$g(\mathbf{x}) = \mathbf{w}^T\mathbf{x} + b = 0.$$

Provided that the dataset is linearly separable, we saw that algorithms such as standard linear classifier and SVM can derive the weights $\mathbf{w}$ and bias b which optimally fit the input data samples. When classes slightly overlap, soft margin SVM is exploited. From this lecture, we will deal with problems that are not linearly separable and for which the design of a linear classifier, even in an optimal way, does not lead to satisfactory performance. The design of nonlinear classifiers emerges now as an inescapable necessity.

To seek non-linearly separable problems one does not need to go into complicated situations. The well-known XOR Boolean function is a typical example of such a problem (see Figure 1). Depending on the values of the input binary data $\mathbf{x} = [x_1, x_2]$, the output is either 0 or 1, and $\mathbf{x}$ is classified into one of the two classes +1 or -1. The corresponding truth table for the XOR operation is shown in Table 1, where a *true* value is expected if the two inputs are not equal and a *false* value if they are equal.

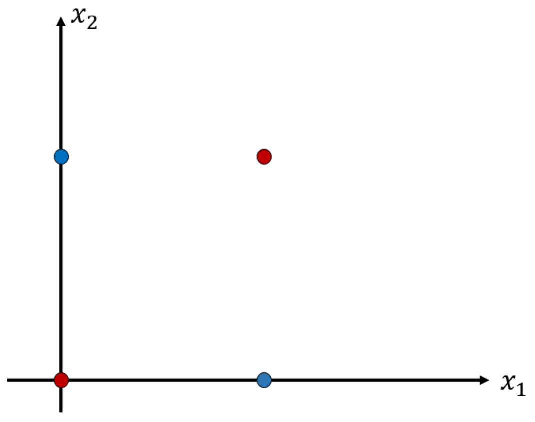

Figure 1: The XOR problem. The blue dots denote *true* and red dots denote *false*.

[*]References
- Christopher Bishop. Pattern Recognition and Machine Learning. 2006
- Sergios Theodoridis, Konstantinos Koutroumbas. Pattern Recognition. 2009
- Understanding Random Forests. March 7, 2022

x_1	0	0	1	1
x_2	0	1	0	1
y	+1	-1	-1	+1

Table 1: Truth labels for the XOR problem.

It is apparent from the XOR problem that no single straight line exists that can separate the two classes, as the decision boundary between the two classes is naturally not linear. Non-linear classifiers are specifically designed to cope with such problems. Common non-linear classifiers include decision trees, random forests, Multi-Layer Perceptron (MLP) algorithm, neural networks, and deep learning. In this lecture, we mainly discuss decision trees and random forests.

2 Decision Trees and Random Forests

2.1 Decision trees

Decision trees are known as multistage decision systems in which classes are sequentially rejected until a finally accepted class is reached. To this end, the feature space is split into unique regions, corresponding to the classes, in a sequential manner. Upon the arrival of a feature vector, the searching of the region to which the feature vector will be assigned is achieved via a sequence of decisions along a path of nodes of an appropriately constructed tree. Such schemes offer advantages when a large number of classes are involved. The most popular decision trees are those that split the space into hyperrectangles with sides parallel to the axes. The sequence of decisions is applied to individual features, and the questions to be answered are of the form

$$\textit{is feature} \quad x_i \leq a \quad ?$$

where a is a threshold value. Such trees are known as Ordinary Binary Classification Trees (OBCTs). Other types of trees are also possible that split the space into convex polyhedral cells or pieces of spheres. Figure 2 gives an example of the OBCT.

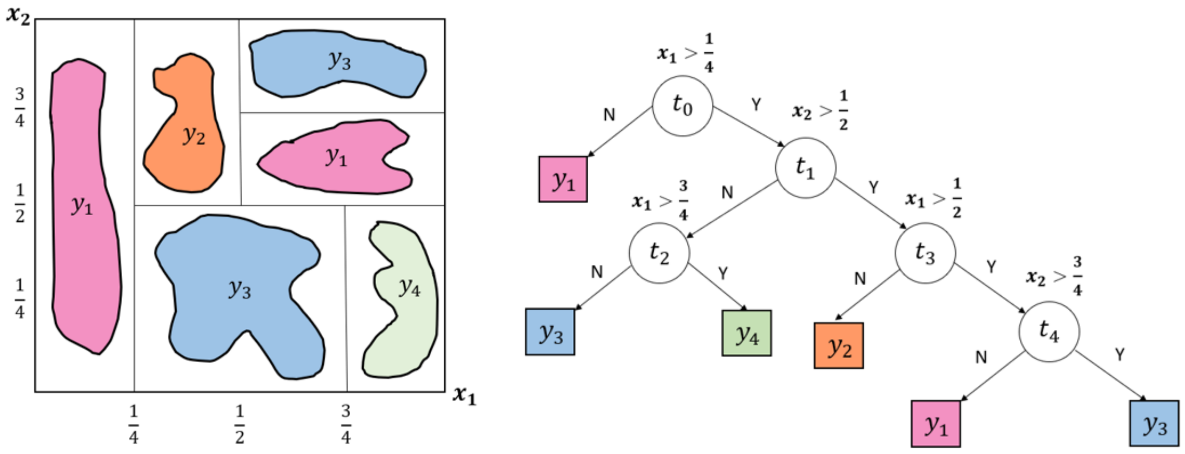

Figure 2: Decision tree classification. (a) OBCT divides the feature space into subregions. (b) A set of questions to be asked for classification.

Splitting criteria. Every binary split of a node generates two descendant nodes. For the tree growing methodology, from the root node down to the leaves, to make sense, every split must generate subsets that are more "class homogeneous" compared to the ancestor's subset. This means that the training feature vectors in each one of the new subsets show a higher preference for specific class(es), whereas data in ancestors are more equally distributed among the classes. The goal, therefore, is to define a measure that quantifies node impurity and split the node so that the overall impurity of the descendant nodes is optimally decreased concerning the ancestor node's impurity.

We consider two different criteria for evaluating the impurity of a given node t: *Gini impurity* and *Entropy impurity*.

- *Gini impurity.* It's given by

$$I(t) = 1 - \sum_{k=1}^{K} p(y_k|t)^2, \tag{1}$$

  where $p(y_k|t)$ is the probability of a vector in note t belongs to the class y_k. K is the total number of classes in the current node. If the node t contains only one class, $I(t) = 0$. Consider a two-class problem, Gini impurity gets its maximum value when the two classes have the same probability, i.e., $I(t) = 1 - (0.5^2 + 0.5^2) = 0.5$.

- *Entropy impurity.* It's given by

$$I(t) = -\sum_{k=1}^{K} p(y_k|t) \cdot \log_2 p(y_k|t), \tag{2}$$

  where log_2 is the logarithm with base 2. This is nothing else than the entropy associated with the data contained in node t, known from Shannon's Information Theory. Similar to Gini, Entropy impurity gets 0 if the node t contains only one class. It gets its maximum value when the probability of the two classes is the same, i.e., $I(t)_{max} = -(0.5 \cdot \log_2 0.5 + 0.5 \cdot \log_2 0.5) = 1$.

Computationally, entropy is more complex since it makes use of logarithms and consequently, the calculation of the Gini index will be faster. In practice, Gini impurity is more widely used (e.g., the default splitting criterion in the scikit-learn library is Gini).

Let us denote the splitted nodes by tY and tN according to the "Yes" or "No" answer to the single question adopted for the node t, also referred as the ancestor node. Let N_t denotes the data contained in t. The descendant nodes are associated with two new subsets, that is, N_{tY}, N_{tN}, respectively. The decrease in node impurity is defined as

$$\Delta I(t) = I(t) - \frac{N_{tY}}{N_t} I(tY) - \frac{N_{tN}}{N_t} I(tN),$$

where $I(tY)$, $I(tN)$ are the impurities of the tY and tN nodes, respectively. The goal is to adopt, from the set of candidate questions, the one that performs the split leading to the highest decrease of impurity.

Stopping rule. The natural question that now arises is when one decides to stop splitting a node and declares it as a leaf of the tree. A possibility is to adopt a threshold T and stop splitting if the maximum value of $\Delta I(t)$ over all possible splits, is less than

T. Other alternatives are to stop splitting either if the subset N_t is small enough or the data in node t is pure (i.e., all data samples in node t belong to the same class).

A critical factor in designing a decision tree is its size. As was the case with the MLPs, the size of a tree must be large enough but not too large; otherwise, it tends to learn the particular details of the training set and exhibits poor generalization performance. Experience has shown that the use of a threshold value for the impurity decreases as the stop-splitting rule does not lead to trees of the right size, because it usually stops tree growing either too early or too late. The most commonly used approach is to grow a tree up to a large size first and then prune nodes according to a pruning criterion.

A drawback associated with tree classifiers is their high variance. In practice, it is common that a small change in the training data set results in a very different tree. The reason for this lies in the hierarchical nature of the tree classifiers. In the next section, random forests are introduced for overcoming such limitations.

2.2 Random forests

A common strategy to overcome the high variance of decision tree classifiers is to combine them. Thus, one can exploit their advantages to reach an overall better performance than could be achieved by using each of them separately. An important observation that justifies such an approach is the following. From the different (candidate) decision trees we design to choose the one that results in the best performance (i.e., the highest classification accuracy). However, different trees may fail (to classify correctly) on different data distributions. That is, even the “best” decision tree can fail on datasets that other trees succeed on. To this end, the random forest algorithm is proposed.

Random forest, like its name implies, consists of a large number of individual decision trees that operate as an ensemble. Each tree in the random forest spits out a class prediction and the class with the most votes becomes the model’s prediction. Figure 3 gives an illustration of the random forest model.

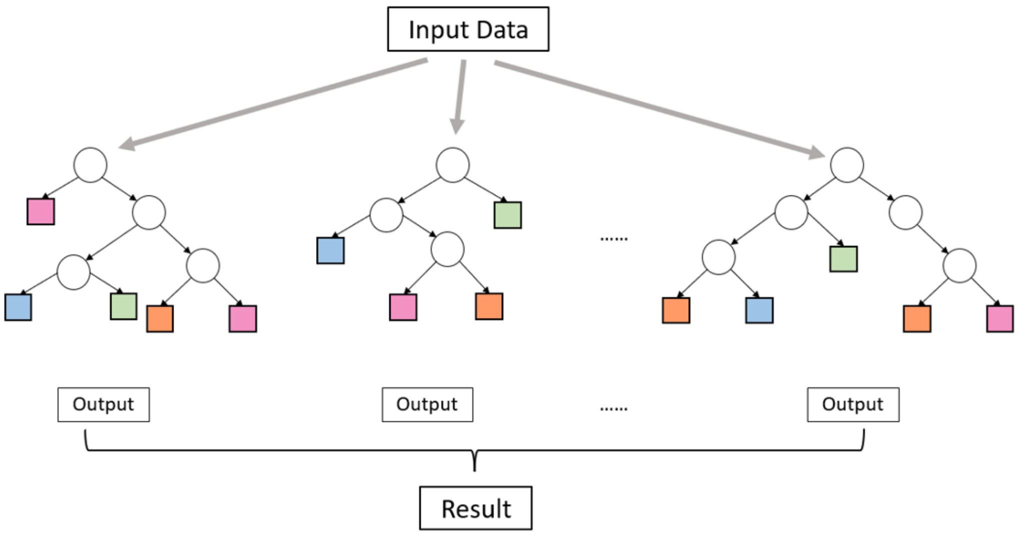

Figure 3: A random forest model.

The fundamental concept behind random forest is a simple but powerful one: the wisdom of crowds. The reason that the random forest model works so well is: *a large* *number of relatively uncorrelated classifiers (trees) operating as a committee will outperform any of the individual constituent classifiers.* Two aspects are involved to ensure the independence among the various tree classifiers in the whole forest:

- **Bagging** (also referred to as *bootstrap aggregation*). It is a technique that can reduce variance and improve the generalization error performance. The basic idea is to create a number (say B) of variants, $X_1, X_2, X_3, ..., X_B$, of the training set, by uniformly sampling from the original dataset X with replacement[1]. For each of the training set variants X_i, a tree T_i is constructed. The final decision is in favor of the class predicted by the majority of the sub-classifiers, $T_i(i = 1, 2, 3, ..., B)$. Note that with replacement, we are not splitting the training data into smaller chunks and training each tree on a different chunk. Rather, if we have a sample of size N, we are still feeding each tree a training set of size N (unless specified otherwise). But instead of the original training data, we take a random sample of size N with a certain level of data repetitiveness. For instance, if our training data was [1, 2, 3, 4, 5, 6] then we might give one of our trees the following list [1, 2, 2, 3, 6, 6]. Notice that both lists are of length six and that "2" and "6" are both repeated in the randomly selected training data we give to our tree (because we sample with replacement).

- **Random feature selection**. In a normal decision tree, when it is time to split a node, we consider every possible feature and pick the one that produces the most separation between the observations in the left node vs. those in the right node using impurity measures. In contrast, each tree in a random forest can pick only from a random subset of features. This forces more variation amongst the trees in the model and ultimately results in lower correlation across trees and more diversification. Figure 4 gives an illustration of random feature selection.

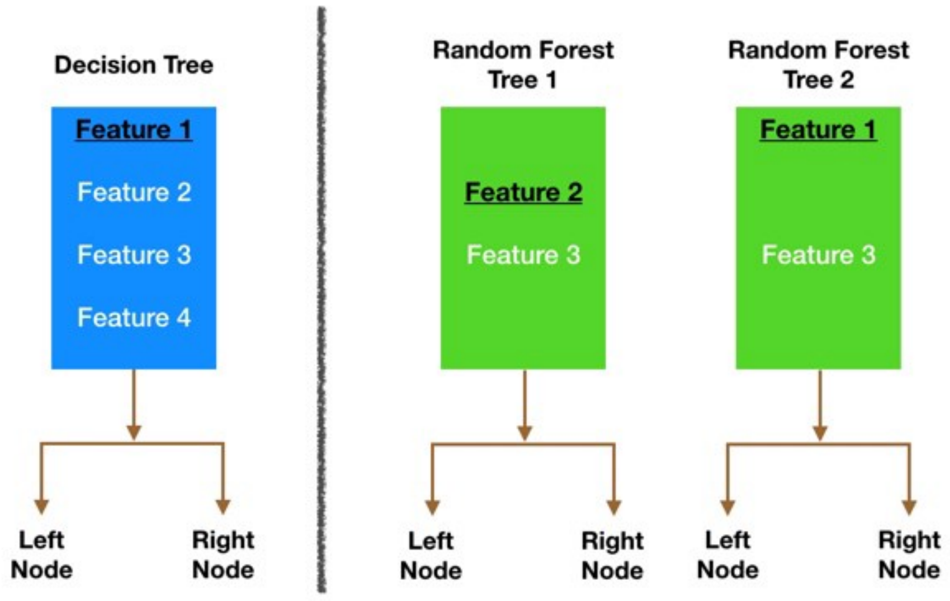

Figure 4: Random feature selection.

[1]It means that we can select the same value multiple times.

Finally, it must be stated that there are close similarities between the decision trees (random forests) and the neural network classifiers. Both aim at forming complex decision boundaries in the feature space. A major difference lies in the way decisions are made. Decision trees (random forests) employ a hierarchically structured decision function sequentially. In contrast, neural networks utilize a set of soft (not final) decisions in a parallel fashion. Furthermore, their training is performed via different philosophies. However, despite their differences, it has been shown that linear tree classifiers (with a linear splitting criterion) can be adequately mapped to an MLP structure. So far, from the performance point of view, comparative studies seem to give an advantage to the MLPs concerning the classification error, and an advantage to the decision trees (random forests) concerning the required training time.

3 Data and Features

3.1 Feature selection

A major problem associated with machine learning is the so-called curse of dimensionality[2]. In practice, it is very common to design a classification system with dozens, or even hundreds of features. There is more than one reason to reduce the number of features to a sufficient minimum. Computational complexity is the obvious one. A related reason is that, although two features may carry good classification information when treated separately, there is little gain if they are combined into a feature vector because of a high mutual correlation. Thus, complexity increases without much gain. Another major reason is imposed by the required generalization properties of the classifier. The higher the ratio of the number of training patterns N to the number of free classifier parameters, the better the generalization properties of the resulting classifier.

A large number of features are directly translated into a large number of classifier parameters (e.g., weights in a linear classifier, synaptic weights in a neural network). Thus, for a finite and usually limited number of training patterns, keeping the number of features as small as possible is in line with our desire to design classifiers with good generalization capabilities. To this end, different strategies will be adopted. One is to select features with sufficient discriminatory capability. An alternative is to use feature reduction techniques such as Principle Component Analysis (PCA). In this section, we focus on the feature selection techniques.

We consider selecting d features from p features in the original feature set. The optimal way of doing it is to brute-force search for all possible combinations of features, evaluate their accuracy over the input dataset, and pick the feature combination that gives the highest performance. However, such a method is usually too computationally expensive. When the feature set and the dataset become large in practice, it's almost impossible to implement such a searching method in a reasonable time. Therefore, we consider the sub-optimal techniques for feature selection.

Instead of the actual model performance (i.e., accuracy), several metrics are introduced to quantitatively measure if a feature is good or not:

[2]Wikipedia: Curse of dimensionality.

- **Within-class scatter matrix**

  $$S_W = \sum_{k=1}^{K} \frac{N_k}{N} \Sigma_k,$$

  where K is the total number of classes. N_k is the number of data samples of the k^{th} class. N is the total number of data samples. Σ_k is the covariance matrix[3] of the data samples of the k^{th} class using a given feature set.

- **Between-class scatter matrix**

  $$S_B = \sum_{k=1}^{K} \frac{N_k}{N} (\mu_{\mathbf{k}} - \mu)(\mu_{\mathbf{k}} - \mu)^T,$$

  where $\mu_{\mathbf{k}}$ is the mean of per-class samples. μ is the mean of all data samples.

Obviously, given the feature set, trace(S_W) measures the average, overall classes, the variance of the features. trace(S_B) measures the average (overall classes) distance of the mean of each class from the respective global value. A good feature set should achieve a small variance within per-class (i.e., small trace(S_W)) and a large distance between classes (i.e., large trace(S_B)). Figure 5 further gives a visualization of S_W and S_B. A common criterion for selecting features is to compute

$$J = \frac{trace(S_B)}{trace(S_W)}.$$

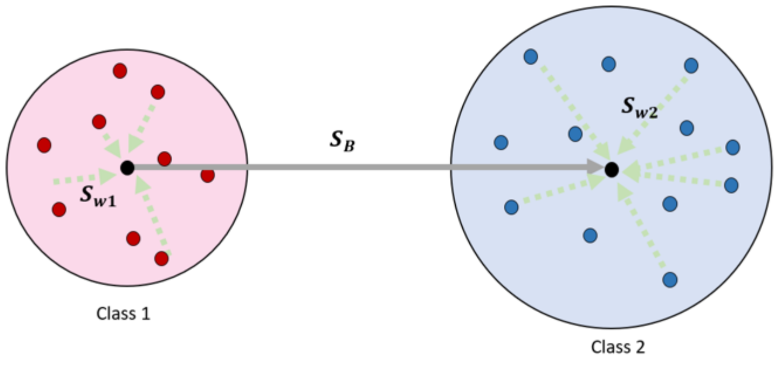

Figure 5: Visualization of scatter matrices.

Using the scatter matrix criterion, we aim to search for the optimal d features from p original features. Instead of brute-force search, several sub-optimal searching algorithms are proposed and used in practice:

1) Compute the criterion value for each of the features. Select the d optimal features with the top values.

2) *Forward search.* Starting from the empty feature set, add one feature each time to the current set which leads to the optimal value according to your criterion, until the d features have been chosen in the set.

3) *Backward search.* Starting from the original feature set, remove one feature each time such that the remaining features have the optimal value of a given criterion, until the $(p - d)$ features have been removed.

[3]Wikipedia: Covariance matrix.

3.2 Classifier evaluation

In previous lectures, we introduced some commonly used metrics for measuring the performance of a classifier on a given dataset (e.g., overall accuracy, mean per-class accuracy, confusion matrix). However, it is not a good idea to measure the performance of a classifier on the training set, as it will generally yield an optimistic estimate of the classifier. Due to the problem of over-fitting (i.e., the classifier performs well on the training set while it gives bad performance when extended to unseen data samples), the performance on the training set is not a good indicator of predictive performance on unseen data.

If data is plentiful, then one approach is simply to use some of the available data (i.e., training set) to train a range of models or a given model with a range of values for its complexity parameters, and then to compare them on independent data (i.e., test set), and select the one having the best predictive performance. In some scenarios, if the model design is too complex and involves too many hyper-parameters, it becomes necessary to keep aside a third validation set on which the effects of various hyper-parameters are evaluated. Then, the performance of the selected model is finally evaluated over the independent test set.

Given a certain amount of data available, how to determine the ratio between the training set and test set?

- Using the same data for training and testing may give a good classifier but will yield an optimistically biased performance estimate.
- Using a large portion of the data set for training will lead to a well-trained classifier. However, correspondingly, a small test set is likely to give an independent yet unreliable (i.e., with large variance) estimate of the model performance.
- Using a large test set instead will give a reliable, unbiased estimate of a badly-trained classifier, due to the small amount of training set.
- In practice, the ratio of the train-test split is usually set to 7:3, 6:4, or 5:5, to achieve a relative balance between these two sets.

In many applications, the supply of data for training and testing will be limited, and to build good models, we wish to use as much of the available data as possible for training. However, we also don't want the test set to be small, as it will give a relatively noisy estimate of predictive performance. One solution to this dilemma is to use *cross-validation*.

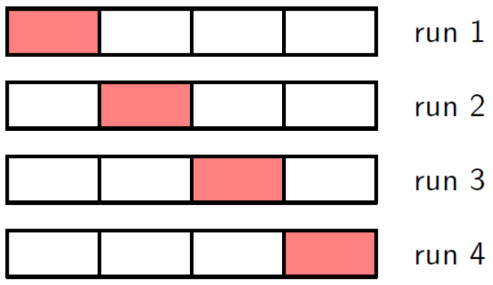

Figure 6: An illustration of K-fold cross validation. In this example $K = 4$. The red block is used for testing while the rest 3 blocks are used for training in each run. The performance score is then averaged over the 4 runs in total.

The technique of K-fold cross-validation, illustrated in Figure 6 for the case of $K = 4$, involves taking the available data and partitioning it into K groups. Then $(K - 1)$ of the groups are used to train a classifier, which is then evaluated on the remaining group. This means a proportion $(K - 1)/K$ of the available data is used for training while making use of all of the rest for testing. The procedure is repeated for all K possible choices for the held-out group, indicated here by the red blocks, and the performance scores from the K runs are then averaged. Sufficient data is crucial for most existing machine learning classifiers, therefore the goal of cross-validation is to allow all the data samples to be seen and trained, while still yielding a reasonable evaluation of the classifier.