Data Analysis and Machine Learning: From Decision Trees to Forests and all that

Morten Hjorth-Jensen[1,2]

[1]Department of Physics, University of Oslo
[2]Department of Physics and Astronomy and National Superconducting Cyclotron Laboratory, Michigan State University

Nov 5, 2019

Decision trees, overarching aims

Decision trees are supervised learning algorithms used for both, classification and regression tasks.

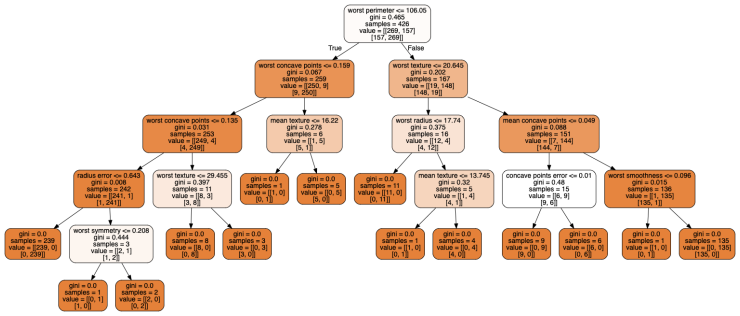
The main idea of decision trees is to find those descriptive features which contain the most **information** regarding the target feature and then split the dataset along the values of these features such that the target feature values for the resulting underlying datasets are as pure as possible.

The descriptive features which reproduce best the target/output features are normally said to be the most informative ones. The process of finding the **most informative** feature is done until we accomplish a stopping criteria where we then finally end up in so called **leaf nodes**.

A decision tree is typically divided into a **root node**, the **interior nodes**, and the final **leaf nodes** or just **leaves**. These entities are then connected by so-called **branches**.

The leaf nodes contain the predictions we will make for new query instances presented to our trained model. This is possible since the model has learned the underlying structure of the training data and hence can, given some assumptions, make predictions about the target feature value (class) of unseen query instances.

A typical Decision Tree with its pertinent Jargon, Classification Problem



This tree was produced using the Wisconsin cancer data (discussed here as well, see code examples below) using **Scikit-Learn**'s decision tree classifier. Here we have used the so-called **gini** index (see below) to split the various branches.

General Features

The overarching approach to decision trees is a top-down approach.

- A leaf provides the classification of a given instance.
- A node specifies a test of some attribute of the instance.
- A branch corresponds to a possible values of an attribute.
- An instance is classified by starting at the root node of the tree, testing the attribute specified by this node, then moving down the tree branch corresponding to the value of the attribute in the given example.

This process is then repeated for the subtree rooted at the new node.

How do we set it up?

In simplified terms, the process of training a decision tree and predicting the target features of query instances is as follows:

1. Present a dataset containing of a number of training instances characterized by a number of descriptive features and a target feature
2. Train the decision tree model by continuously splitting the target feature along the values of the descriptive features using a measure of information gain during the training process
3. Grow the tree until we accomplish a stopping criteria create leaf nodes which represent the *predictions* we want to make for new query instances

4. Show query instances to the tree and run down the tree until we arrive at leaf nodes

Then we are essentially done!

Decision trees and Regression

Building a tree, regression

There are mainly two steps

1. We split the predictor space (the set of possible values $x_1, x_2, \ldots, x_p$) into J distinct and non-non-overlapping regions, $R_1, R_2, \ldots, R_J$.

2. For every observation that falls into the region R_j , we make the same prediction, which is simply the mean of the response values for the training observations in R_j.

How do we construct the regions $R_1, \ldots, R_J$? In theory, the regions could have any shape. However, we choose to divide the predictor space into high-dimensional rectangles, or boxes, for simplicity and for ease of interpretation of the resulting predictive model. The goal is to find boxes $R_1, \ldots, R_J$ that minimize the MSE, given by

$$\sum_{j=1}^{J} \sum_{i \in R_j} (y_i - \overline{y}_{R_j})^2,$$

where $\overline{y}_{R_j}$ is the mean response for the training observations within box j.

A top-down approach, recursive binary splitting

Unfortunately, it is computationally infeasible to consider every possible partition of the feature space into J boxes. The common strategy is to take a top-down approach

The approach is top-down because it begins at the top of the tree (all observations belong to a single region) and then successively splits the predictor space; each split is indicated via two new branches further down on the tree. It is greedy because at each step of the tree-building process, the best split is made at that particular step, rather than looking ahead and picking a split that will lead to a better tree in some future step.

Making a tree

In order to implement the recursive binary splitting we start by selecting the predictor x_j and a cutpoint s that splits the predictor space into two regions R_1 and R_2

$$\{X | x_j < s\},$$

and

$$\{X|x_j \geq s\},$$

so that we obtain the lowest MSE, that is

$$\sum_{i:x_i \in R_j} (y_i - \overline{y}_{R_1})^2 + \sum_{i:x_i \in R_2} (y_i - \overline{y}_{R_2})^2,$$

which we want to minimize by considering all predictors $x_1, x_2, \ldots, x_p$. We consider also all possible values of s for each predictor. These values could be determined by randomly assigned numbers or by starting at the midpoint and then proceed till we find an optimal value.

For any j and s, we define the pair of half-planes where $\overline{y}_{R_1}$ is the mean response for the training observations in $R_1(j, s)$, and $\overline{y}_{R_2}$ is the mean response for the training observations in $R_2(j, s)$.

Finding the values of j and s that minimize the above equation can be done quite quickly, especially when the number of features p is not too large.

Next, we repeat the process, looking for the best predictor and best cutpoint in order to split the data further so as to minimize the MSE within each of the resulting regions. However, this time, instead of splitting the entire predictor space, we split one of the two previously identified regions. We now have three regions. Again, we look to split one of these three regions further, so as to minimize the MSE. The process continues until a stopping criterion is reached; for instance, we may continue until no region contains more than five observations.

Pruning the tree

The above procedure is rather straightforward, but leads often to overfitting and unnecessarily large and complicated trees. The basic idea is to grow a large tree T_0 and then prune it back in order to obtain a subtree. A smaller tree with fewer splits (fewer regions) can lead to smaller variance and better interpretation at the cost of a little more bias.

The so-called Cost complexity pruning algorithm gives us a way to do just this. Rather than considering every possible subtree, we consider a sequence of trees indexed by a nonnegative tuning parameter α.

Cost complexity pruning

For each value of α there corresponds a subtree $T \in T_0$ such that

$$\sum_{m=1}^{\overline{T}} \sum_{i:x_i \in R_m} (y_i - \overline{y}_{R_m})^2 + \alpha \overline{T},$$

is as small as possible. Here $\overline{T}$ is the number of terminal nodes of the tree T , R_m is the rectangle (i.e. the subset of predictor space) corresponding to the m-th terminal node.

The tuning parameter α controls a trade-off between the subtree's complexity and its fit to the training data. When $\alpha = 0$, then the subtree T will simply equal T_0, because then the above equation just measures the training error. However, as α increases, there is a price to pay for having a tree with many terminal nodes. The above equation will tend to be minimized for a smaller subtree.

It turns out that as we increase α from zero branches get pruned from the tree in a nested and predictable fashion, so obtaining the whole sequence of subtrees as a function of α is easy. We can select a value of α using a validation set or using cross-validation. We then return to the full data set and obtain the subtree corresponding to α.

Schematic Regression Procedure

Building a Regression Tree.

1. Use recursive binary splitting to grow a large tree on the training data, stopping only when each terminal node has fewer than some minimum number of observations.

2. Apply cost complexity pruning to the large tree in order to obtain a sequence of best subtrees, as a function of α.

3. Use for example K-fold cross-validation to choose α. Divide the training observations into K folds. For each $k = 1, 2, \ldots, K$ we:

   - repeat steps 1 and 2 on all but the k-th fold of the training data.
   - Then we valuate the mean squared prediction error on the data in the left-out k-th fold, as a function of α.
   - Finally we average the results for each value of α, and pick α to minimize the average error.

4. Return the subtree from Step 2 that corresponds to the chosen value of α.

A Classification Tree

A classification tree is very similar to a regression tree, except that it is used to predict a qualitative response rather than a quantitative one. Recall that for a regression tree, the predicted response for an observation is given by the mean response of the training observations that belong to the same terminal node. In contrast, for a classification tree, we predict that each observation belongs to the most commonly occurring class of training observations in the region to which it belongs. In interpreting the results of a classification tree, we are often interested not only in the class prediction corresponding to a particular terminal node region, but also in the class proportions among the training observations that fall into that region.

Growing a classification tree

The task of growing a classification tree is quite similar to the task of growing a regression tree. Just as in the regression setting, we use recursive binary splitting to grow a classification tree. However, in the classification setting, the MSE cannot be used as a criterion for making the binary splits. A natural alternative to MSE is the **classification error rate**. Since we plan to assign an observation in a given region to the most commonly occurring error rate class of training observations in that region, the classification error rate is simply the fraction of the training observations in that region that do not belong to the most common class.

When building a classification tree, either the Gini index or the entropy are typically used to evaluate the quality of a particular split, since these two approaches are more sensitive to node purity than is the classification error rate.

Classification tree, how to split nodes

If our targets are the outcome of a classification process that takes for example $k = 1, 2, \dots, K$ values, the only thing we need to think of is to set up the splitting criteria for each node.

We define a PDF p_{mk} that represents the number of observations of a class k in a region R_m with N_m observations. We represent this likelihood function in terms of the proportion $I(y_i = k)$ of observations of this class in the region R_m as

$$p_{mk} = \frac{1}{N_m} \sum_{x_i \in R_m} I(y_i = k).$$

We let p_{mk} represent the majority class of observations in region m. The three most common ways of splitting a node are given by

- Misclassification error

$$p_{mk} = \frac{1}{N_m} \sum_{x_i \in R_m} I(y_i \neq k) = 1 - p_{mk}.$$

- Gini index g

$$g = \sum_{k=1}^{K} p_{mk}(1 - p_{mk}).$$

- Information entropy or just entropy s

$$s = -\sum_{k=1}^{K} p_{mk} \log p_{mk}.$$

Visualizing the Tree, Classification

Visualizing the Tree, The Moons

Algorithms for Setting up Decision Trees

Two algorithms stand out in the set up of decision trees:

1. The CART (Classification And Regression Tree) algorithm for both classification and regression
2. The ID3 algorithm based on the computation of the information gain for classification

We discuss both algorithms with applications here. The popular library -Scikit-Learn$_u sestheCARTalgorithm.Forclassificationproblemsyoucanuseeitherthe$**gini**$indexorthe$**entropy**$tosplito$

The CART algorithm for Classification

The CART algorithm for Regression

Computing the Gini index

The example we will look at is a classical one in many Machine Learning applications. Based on various meteorological features, we have several so-called attributes which decide whether we at the end will do some outdoor activity like skiing, going for a bike ride etc etc. The table here contains the feautures **outlook**, **temperature**, **humidity** and **wind**. The target or output is whether we ride (True=1) or whether we do something else that day (False=0). The attributes for each feature are then sunny, overcast and rain for the outlook, hot, cold and mild for temperature, high and normal for humidity and weak and strong for wind.

The table here summarizes the various attributes and

Day	Outlook	Temperature	Humidity	Wind	Ride
1	Sunny	Hot	High	Weak	0
2	Sunny	Hot	High	Strong	1
3	Overcast	Hot	High	Weak	1
4	Rain	Mild	High	Weak	1
5	Rain	Cool	Normal	Weak	1
6	Rain	Cool	Normal	Strong	0
7	Overcast	Cool	Normal	Strong	1
8	Sunny	Mild	High	Weak	0
9	Sunny	Cool	Normal	Weak	1
10	Rain	Mild	Normal	Weak	1
11	Sunny	Mild	Normal	Strong	1
12	Overcast	Mild	High	Strong	1
13	Overcast	Hot	Normal	Weak	1
14	Rain	Mild	High	Strong	0

Simple Python Code to read in Data and perform Classification

Computing the Gini Factor

The above functions (gini, entropy and misclassification error) are important components of the so-called CART algorithm. We will discuss this algorithm below after we have discussed the information gain algorithm ID3.

In the example here we have converted all our attributes into numerical values $0, 1, 2$ etc.

Entropy and the ID3 algorithm

ID3, learns decision trees by constructing them topdown, beginning with the question **which attribute should be tested at the root of the tree**?

1. Each instance attribute is evaluated using a statistical test to determine how well it alone classifies the training examples.
2. The best attribute is selected and used as the test at the root node of the tree.
3. A descendant of the root node is then created for each possible value of this attribute.
4. Training examples are sorted to the appropriate descendant node.
5. The entire process is then repeated using the training examples associated with each descendant node to select the best attribute to test at that point in the tree.
6. This forms a greedy search for an acceptable decision tree, in which the algorithm never backtracks to reconsider earlier choices.

The ID3 algorithm selects, which attribute to test at each node in the tree.

We would like to select the attribute that is most useful for classifying examples.

What is a good quantitative measure of the worth of an attribute?

Information gain measures how well a given attribute separates the training examples according to their target classification.

The ID3 algorithm uses this information gain measure to select among the candidate attributes at each step while growing the tree.

Implementing the ID3 Algorithm

Cancer Data again now with Decision Trees and other Methods

Another example, the moons again

Playing around with regions

Regression trees

Final regressor code

Pros and cons of trees, pros

- White box, easy to interpret model. Some people believe that decision trees more closely mirror human decision-making than do the regression and classification approaches discussed earlier (think of support vector machines)
- Trees are very easy to explain to people. In fact, they are even easier to explain than linear regression!
- No feature normalization needed
- Tree models can handle both continuous and categorical data (Classification and Regression Trees)
- Can model nonlinear relationships
- Can model interactions between the different descriptive features
- Trees can be displayed graphically, and are easily interpreted even by a non-expert (especially if they are small)

Disadvantages

- Unfortunately, trees generally do not have the same level of predictive accuracy as some of the other regression and classification approaches
- If continuous features are used the tree may become quite large and hence less interpretable
- Decision trees are prone to overfit the training data and hence do not well generalize the data if no stopping criteria or improvements like pruning, boosting or bagging are implemented
- Small changes in the data may lead to a completely different tree. This issue can be addressed by using ensemble methods like bagging, boosting or random forests

- Unbalanced datasets where some target feature values occur much more frequently than others may lead to biased trees since the frequently occurring feature values are preferred over the less frequently occurring ones.
- If the number of features is relatively large (high dimensional) and the number of instances is relatively low, the tree might overfit the data
- Features with many levels may be preferred over features with less levels since for them it is *more easy* to split the dataset such that the sub datasets only contain pure target feature values. This issue can be addressed by preferring for instance the information gain ratio as splitting criteria over information gain

However, by aggregating many decision trees, using methods like bagging, random forests, and boosting, the predictive performance of trees can be substantially improved.

From a Single Tree to Many Trees, Meet the Jungle of Methods

As stated above and seen in many of the examples discussed here about a single decision tree, we often end up overfitting our training data. This normally means that we have a high variance. Can we reduce the variance of a statistical learning method?

This leads us to a set of different methods that can combine different machine learning algorithms or just use one of them to construct forests and jungles of trees, homogeneous ones or heterogenous ones. These methods are recognized by different names which we will try to explain here. These are

1. Voting classifiers
2. Bagging and Pasting
3. Random forests
4. Boosting methods

We discuss these methods here.

Bagging

The **plain** decision trees suffer from high variance. This means that if we split the training data into two parts at random, and fit a decision tree to both halves, the results that we get could be quite different. In contrast, a procedure with low variance will yield similar results if applied repeatedly to distinct data sets; linear regression tends to have low variance, if the ratio of n to p is moderately large.

Bootstrap aggregation, or just **bagging**, is a general-purpose procedure for reducing the variance of a statistical learning method.

More bagging

Bagging typically results in improved accuracy over prediction using a single tree. Unfortunately, however, it can be difficult to interpret the resulting model. Recall that one of the advantages of decision trees is the attractive and easily interpreted diagram that results.

However, when we bag a large number of trees, it is no longer possible to represent the resulting statistical learning procedure using a single tree, and it is no longer clear which variables are most important to the procedure. Thus, bagging improves prediction accuracy at the expense of interpretability. Although the collection of bagged trees is much more difficult to interpret than a single tree, one can obtain an overall summary of the importance of each predictor using the MSE (for bagging regression trees) or the Gini index (for bagging classification trees). In the case of bagging regression trees, we can record the total amount that the MSE is decreased due to splits over a given predictor, averaged over all B possible trees. A large value indicates an important predictor. Similarly, in the context of bagging classification trees, we can add up the total amount that the Gini index is decreased by splits over a given predictor, averaged over all B trees.

Simple Voting Example, head or tail

Using the Voting Classifier

Please, not the moons again! Voting and Bagging

Now Bagging

Random forests

Random forests provide an improvement over bagged trees by way of a small tweak that decorrelates the trees.

As in bagging, we build a number of decision trees on bootstrapped training samples. But when building these decision trees, each time a split in a tree is considered, a random sample of m predictors is chosen as split candidates from the full set of p predictors. The split is allowed to use only one of those m predictors.

A fresh sample of m predictors is taken at each split, and typically we choose

$$m \approx \sqrt{p}.$$

In building a random forest, at each split in the tree, the algorithm is not even allowed to consider a majority of the available predictors.

The reason for this is rather clever. Suppose that there is one very strong predictor in the data set, along with a number of other moderately strong predictors. Then in the collection of bagged variable importance random forest trees, most or all of the trees will use this strong predictor in the top split. Consequently, all of the bagged trees will look quite similar to each other. Hence

the predictions from the bagged trees will be highly correlated. Unfortunately, averaging many highly correlated quantities does not lead to as large of a reduction in variance as averaging many uncorrelated quanti- ties. In particular, this means that bagging will not lead to a substantial reduction in variance over a single tree in this setting.

Random Forest Algorithm

The algorithm described here can be applied to both classification and regression problems.

We will grow of forest of say M trees.

1. For $m = 1 : M$ we
   - Draw a bootstrap sample of from the training data organized in our $\boldsymbol{X}$ matrix.
   - We grow then a random forest tree T_m based on the bootstrapped data by repeating the steps outlined till we reach the maximum node size is reached
     (a) we select $m \leq p$ varibales at random from the p predictors/features
     (b) pick the best split point among the m features using either the CART algorithm or the ID3 for classification and create a new node
     (c) split the node into daughter nodes
2. Output then the ensemble of trees $\{T_m\}_1^M$ and make predictions for either a regression type of problem or a classification type of problem.

Random Forests Compared with other Methods on the Cancer Data

Compare Bagging on Trees with Random Forests

Feature Importance

Example will be added here.

Boosting, a Bird'e Eye

The basic idea is to combine weak classifiers in order to create a good classifier. With a weak classifier we often intend a classifier which produces results which are only slightly better than we would get by random guesses.

This is done by applying in an iterative way a weak (or a standard classifier like decision trees) to modify the data. In each iteration we emphasize those observations which are misclassified by weighting them with a factor.

Adaptive boosting: AdaBoost, Basic Algorithm

The algorithm here is rather straightforward. Assume that our weak classifier is a decision tree and we consider a binary set of outputs with $y_i \in \{-1, 1\}$ and $i = 0, 1, 2, \ldots, n-1$ as our set of observations. Our design matrix is given in terms of the feature/predictor vectors $\boldsymbol{X} = [\boldsymbol{x}_0\boldsymbol{x}_1 x_{p-1}$. Finally, we define also a classifier determined by our data via a function $G(\boldsymbol{X})$. This function tells us how well we are able to classify our outputs/targets $\boldsymbol{y}$.

We can then define the misclassification error err as

$$\text{err} = \frac{1}{n}\sum_{i=0}^{n-1} I(y_i \neq G(\boldsymbol{X}_{i*}),$$

where the function $I()$ is one if we misclassify and zero if we classify correctly.

Basic Steps of AdaBoost

With the above definitions we are now ready to set up the algorithm for AdaBoost. The basic idea is to set up weights which will be used to scale the correctly classified and the misclassified cases.

1. We start by initializing all weights to $w_i = 1/n$, with $i = 0, 1, 2, \ldots n-1$. It is to see then that $\sum_{i=0}^{n-1} w_i = 1$.

2. We rewrite the misclassification error as

$$\text{err} = \frac{\sum_{i=0}^{n-1} w_i I(y_i \neq G(\boldsymbol{X}_{i*})}{\sum_{i=0}^{n-1} w_i},$$

1. Then we start looping over all attempts at classifying, namely we start an iterative process for $m = 1 : M$, where M is the final number of classifications. Our given classifier could for example be a plain decision tree.

   (a) Fit thus a given classifier to the training using the weights w_i.

   (b) Compute then err and figure out which events are classified properly and which are classified wrongly.

   (c) Define a quantity $\alpha_m = \log(1 - \text{err})/\text{err}$

(d) Set the new weights to $w_i = w_i \times \exp(\alpha_m I(y_i \neq G(\boldsymbol{X}_{i*})$.

2. Compute the new classifier $G(\boldsymbol{X}) = \sum_{i=0}^{n-1} \alpha_m I(y_i \neq G(\boldsymbol{X}_{i*})$.
For the iterations with $m \leq 2$ the weights are modified individually at each steps. The obersvations which were misclassified at iteration $m-1$ have a weight which is larger than those which were classified properly. As this proceeds, the observations which were difficult to classifiy correctly are given a larger influence. Each new classificatio step m is then forced to concentrate on those observations that are missed in the previous iterations.

Figure to Illustrate the Iterative Classification Process

AdaBoost Examples

Using **Scikit-Learn** it is easy to appply the adaptive boosting algorithm, as done here.

Gradient boosting: Basics

Gradient boosting is again a similar technique to Adapative boosting, it combines so-called weak classifiers or regressors into a strong method via a series of iterations.

In order to understand the method, let us illustrate its basics by bringing back the essential steps in linear regression, where our cost function was the least squares function.

Gradient Boosting, algorithm

Suppose we have a cost function $C(f) = \sum_{i=0}^{n-1} L(y_i, f(x_i))$ where y_i is our target and $f(x_i)$ the function which is meant to model y_i. The above cost function could be our standard least squares function

$$C(\boldsymbol{y}, \boldsymbol{f}) = \frac{1}{n}\sum_{i=0}^{n-1}(y_i - f(x_i))^2.$$

The way we proceed in an iterative fashion is to

1. Initialize our estimate by $f_0(x) = 0$.
2. For $m = 1 : M$, we
   (a) compute the negative gradient vector $\boldsymbol{u}_m = -\partial C(\boldsymbol{y}, \boldsymbol{f})/\partial \boldsymbol{f}(x)$ at $f(x) = f_{m-1}(x)$;
   (b) fit the so-called base-learner to the negative gradient $h_m(u_m, x)$;
   (c) update the estimate $f_m(x) = f_{m-1}(x) + \nu h_m(u_m, x)$;
3. The final estimate is then $f_M(x) = \sum_{m=1}^{M} \nu h_m(u_m, x)$.

Gradient Boosting, Examples

Gradient Boots with Early Stopping

XGBoost: Extreme Gradient Boosting

XGBoost or Extreme Gradient Boosting, is an optimized distributed gradient boosting library designed to be highly efficient, flexible and portable. It implements machine learning algorithms under the Gradient Boosting framework. XGBoost provides a parallel tree boosting that solve many data science problems in a fast and accurate way