Decision trees: some additional notes

Richard Johansson

1 Introduction

Decision trees are among the most intuitively understandable models in machine learning. We first give a recap of decision trees in the setup we saw in the lecture: we want to learn *classifiers*: the output variable is discrete. We also restricted ourselves to the case where the features are discrete-valued. After this recap, we then see how we can go beyond these limitations, so that we can consider numerical features or output variables (that is, *regression* instead of classification).

In practice, a decision tree is less useful on its own than in an *ensemble model*: that is, a collection of several trees. For instance, *random forests* are popular tree-based ensembles for classification and regression. Boosting, and in particular *AdaBoost* and *gradient boosting*, are also important types of ensembles that often use decision trees as the base model. These types of tree-based ensemble predictors have achieved state-of-the-art results for many tasks and will be covered later in the course, when we discuss ensembles.

2 Decision trees for classification

Decision tree classifiers classify an input x by answering a few questions about the features in x. (Think "20 questions".) Or, we can view the decision tree as a set of nested `if...then` statements. Decision trees are typically drawn as hierarchical trees. To exemplify, Figure 1 shows a decision tree for the artificially simple task of predicting whether a given person owns a car or not. The classifier uses three discrete-valued features: *residence*, which can be *city* or *country*; *education*, which can be *university* or *other*; and *gender*, which can be *male* or *female*.

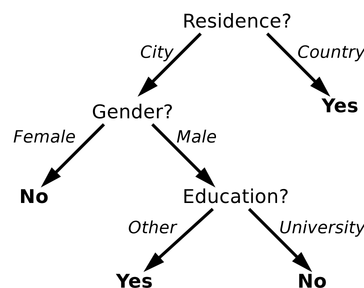


Figure 1: Example of a decision tree for a classification task.

To carry out a prediction for a given individual, we start at the *root node*, which in this case will consider the *residence*. If the residence for this individual is *country*, we reach a *leaf node*, meaning that we are ready to return our prediction, in this case *Yes*. Otherwise, we proceed down to another branch node and consider the *gender*, and so on.

Decision tree training is typically done by finding the most "useful" feature and assigning it to be the top of the tree. Intuitively, a "useful" feature splits the training set into subsets that are more homogeneous than the original set. Figure 2 exemplifies this idea. Here, the feature *education* is more useful than *gender*, because the split by education gives us two subsets (university-educated people and others) that are more homogeneous than the split by gender (into males and females). After splitting into subsets, new decision trees are trained for the subsets (the algorithm is *recursive*).

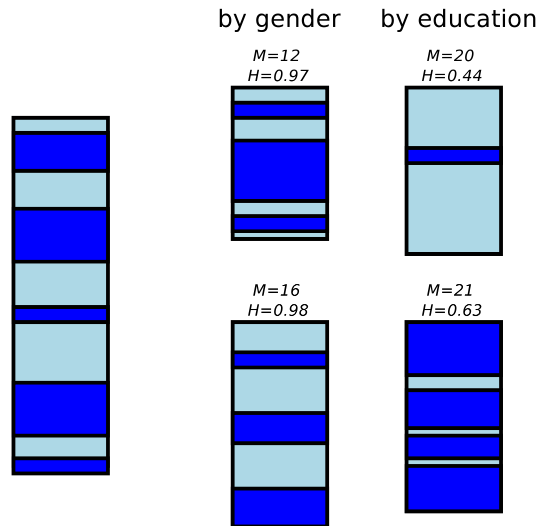


Figure 2: Example of splitting a dataset by two different features.

The notion of a set being homogeneous can be measured in a few different ways. We'll discuss this in §2.2 below.

2.1 Pseudocode for the learning algorithm for decision tree classifiers

This is a recursive algorithm with two *base cases* where the recursion is terminated:

- If we are considering a subset where the output variable has only one value (the set is completely homogeneous), we return a tree leaf with that value.
- If we have reached the maximally allowed tree depth, we return a tree leaf with the most common value among the output values in the dataset.

If we don't end up in one of the base cases, we come to the recursive step, where we pick one feature, create subsets based on the possible values of that feature, and then build subtrees for each of the subsets. Below is the pseudocode for full algorithm, as given in the lecture.

```
def TrainDecisionTreeClassifier(X, Y)
    if all outputs in Y are identical
        return a leaf with the class of the examples in Y
    if we have reached the maximally allowed depth
        return a leaf with the majority class of Y
    F ← MostUsefulFeature(X, Y)
    for each possible value f_i of F
        X_i, Y_i ← the subset where F = f_i
        tree_i ← TrainDecisionTreeClassifier(X_i, Y_i)
    return a tree node that splits on F,
        where f_i is connected to the subtree tree_i
```

2.2 Feature ranking criteria for selecting the "most useful feature"

How do we actually implement the function MostUsefulFeature? We will see a few different alternatives. See also the description in Wikipedia for a description of these criteria.

Our running example will be the situation in Figure 2, where we can split the original training set either by gender or by education level. To exemplify, let's say that the output

variable describes whether the person has a car or not. In this training set, there are 47 individuals, out of which 23 are car owners. Let's consider the two features:

- Gender = Male: 12 owners, 8 non-owners. Gender = Female: 11 owners, 16 non-owners.
- Education level = University: 2 owners, 20 non-owners. Education level = Other: 21 owners, 4 non-owners.

Majority-class counting. In the course book, a simple ranking criterion is used: we simply count the most frequent output class in each subset. In our case, we would then prefer to split by education level, since this gives us majority-class count of 20+21, while splitting by gender would give use 12+16.

Information gain. The ID3 algorithm (Quinlan, 1986), one of the classical implementations of decision tree learning, builds on the notion of using the *entropy* to measure the diversity of a subset. As you may recall from information theory, the entropy of a discrete probability distribution is defined

$$H(Y) = -\sum_i p_i \cdot \log_2 p_i$$

where p_i is the probability of the value i, and the sum goes over all possible values of the output variable Y.[1] The entropy is minimized (it is zero) for a distribution that is completely homogeneous: that is, one of the p_i is equal to one and the rest of them are zero. It is maximized if the distribution is uniform (all the p_i are equal).

To use the entropy to measure the diversity of the output classes in a training set, we first need to convert the counts into probabilities. For instance, in the original training set Y mentioned above, we have the counts 23 and 24, so we get

$$H(Y) = -\frac{23}{47} \cdot \log_2 \frac{23}{47} - \frac{24}{47} \cdot \log_2 \frac{24}{47} = 0.9997$$

When training decision tree classifiers, we look at the entropy of the subsets that we get by splitting on some feature. The key idea is that we prefer splits that lead to a large reduction of entropy. This reduction is called the *information gain*. The complete formula, assuming that the feature F can take m different values, is

$$\text{information-gain}(Y, F) = H(Y) - \sum_{i=1}^{m} \frac{|Y_i|}{|Y|} H(Y_i)$$

As in the pseudocode above, Y_i means the subset of the complete dataset that we get by considering one value of the feature F, such as the female subset when considering the gender. The notation $|Y_i|$ means the size of the subset Y_i.

Let's go back to the example. If we split by gender, we have the entropies 0.9710 and 0.9751 in the male and female subsets, respectively. There are 20 males and 27 females. So we get

$$\text{IG}(Y, \text{gender}) = 0.9997 - (\frac{20}{47} \cdot 0.9710 + \frac{27}{47} \cdot 0.9751) = 0.026.$$

The information gain we would get by splitting by the education level would be higher, so again we would select this feature.

[1] The quantity $p_i \cdot \log_2 p_i$ is strictly speaking undefined if $p_i = 0$, but by convention it is said to be equal to 0 when computing an entropy.

Gini impurity. The CART algorithm (Breiman et al., 1984) used the Gini impurity, which is an alternative to the entropy. Using the same notation as above, the Gini impurity is defined

$$I_G(Y) = 1 - \sum_i p_i^2$$

Just like the entropy, this quantity is equal to zero if the distribution is homogeneous and maximized if it is uniform. This is the default feature ranker used with decision tree classifiers in scikit-learn.

Gain ratio. The C4.5 algorithm (Quinlan, 1993), an extension of ID3, used a modified version of the information gain called the *gain ratio*. The information gain measure has a drawback that it tends to give preference to features that has a large number of possible values, so the gain ratio tries to counterbalance this by introducing a factor that downweights those features.

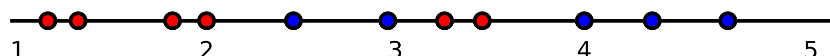
2.3 Decision tree classifiers with numerical features

When we had discrete-valued features, such as *education* or *gender* in the previous example, the possible values of a feature divide the training set into subsets. For instance, the feature *education* splits the training set into university-educated people and others.

For a numerical feature (such as *income*, given in some currency), the most common approach is to select a *threshold* and split the dataset into two subgroups: above and below that threshold. This threshold is selected so that it optimizes the split quality score, such as the majority count, entropy, or Gini impurity discussed above.

Which possible threshold values do we consider? Naively, we could consider a brute-force approach, considering all values between the lowest and the highest observed value, with some small step (e.g. 1.00, 1.01, . . . , 9.99, 10.00). This is going to be much too inefficient. A better approach would sort all the numerical values and consider a point between every pair of adjacent values.

To exemplify, let's say that we'd like to determine whether a person owns a car, based on the income level. We sort all the people by the income level. For every individual, with an income level x, we set the threshold at x and compute our selected homogeneity criterion based on this split. In the end, we select the threshold that gives us the most homogeneous split. We might have a situation as in the following figure; here, let's say that the blue dots represent people who own a car and red dots people who don't, and the x axis represents the income level in some currency.



You can verify that we get the best homogeneity score, measured by the information gain or Gini impurity, if we put the threshold somewhere between 2 and 2.5. If we measure the homogeneity by majority-class counting, a threshold between 3.5 and 4 would be considered equally good.

To summarize, we get an algorithm as in the pseudocode below.

def FindBestSplit(X, Y, f)
  sort the dataset X, Y by the feature f in ascending order
  for every instance x_i, y_i
    let x_i^f be the value of the feature f for the instance x_i
    compute the homogeneity criterion based on a threshold at x_i^f
  return the threshold that maximizes the homogeneity criterion

2.4 Visualizing the decision boundaries for decision tree classifiers

In simple datasets where we just have two features, we can plot the boundary between classes defined by a classifier: its *decision boundary*. For decision trees with numerical features, this boundary has a characteristic shape consisting of straight line segments that run parallel to the x and y axes. The reason why the boundaries have this shape is that the decision trees consider one feature at a time,

Figure 3 shows the decision boundaries for two different tree classifiers trained on the iris dataset. When the tree is deeper, the decision boundary will have a more complex and "jagged" shape, as in the second of the two trees in the figure.

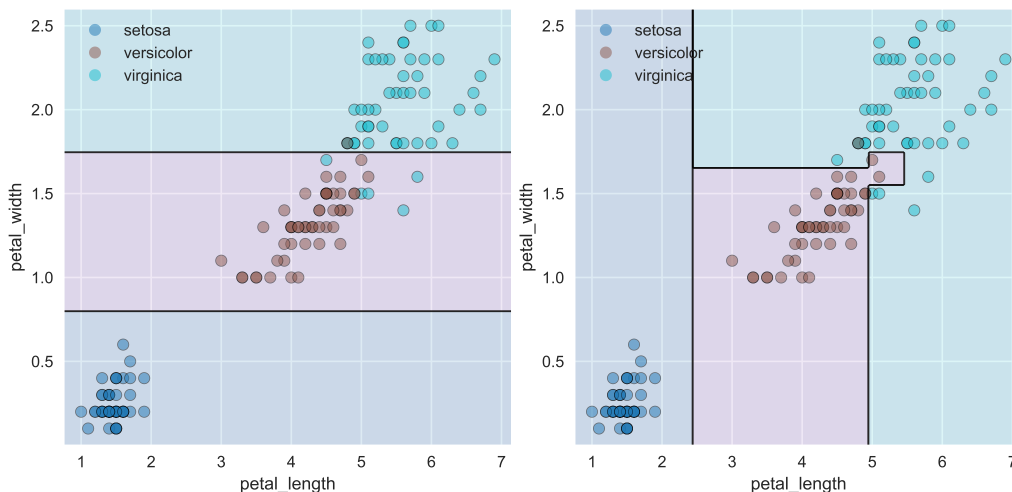


Figure 3: Decision boundaries for two decision tree classifiers for the iris dataset.

2.5 Overfitting and underfitting in decision trees

Decision trees are very expressive and they can easily "memorize" the data in a training set. For this reason, a weakness of decision trees is that they have a tendency to *overfit*: learn over-complicated models that generalize poorly to new data.

There are some ways we can try to mitigate the risk of overfitting. Most obviously, we can impose various types of restrictions to keep the tree small. For instance, in the algorithm presented above, we had a special base case that terminates the recursion if the tree has reached the maximally allowed tree depth. Figure 4 shows the accuracy on the training set and test set for decision tree classifier, as a function of the maximally allowed tree depth. When the trees grow deeper than 7 levels, the test set accuracy starts to decrease, which means that we should probably set the maximal depth to 6 in this case.

In scikit-learn's `DecisionTreeClassifier` and `DecisionTreeRegressor`, we can control the maximal depth using the hyperparameter `max_depth`. In scikit-learn, we can also use a restriction on the number of nodes instead of the depth. A smarter alternative, not available in scikit-learn, is to apply *tree pruning*, where a node is removed from a tree if its removal does not lead to a drop in accuracy on some validation set.

Conversely, we can run into the problem of *underfitting* if we are too restrictive with the tree depth or prune too aggressively. This means that the model is too simple to capture the complexitites of the dataset. In Figure 4, we see underfitting when the depth is less than 6.

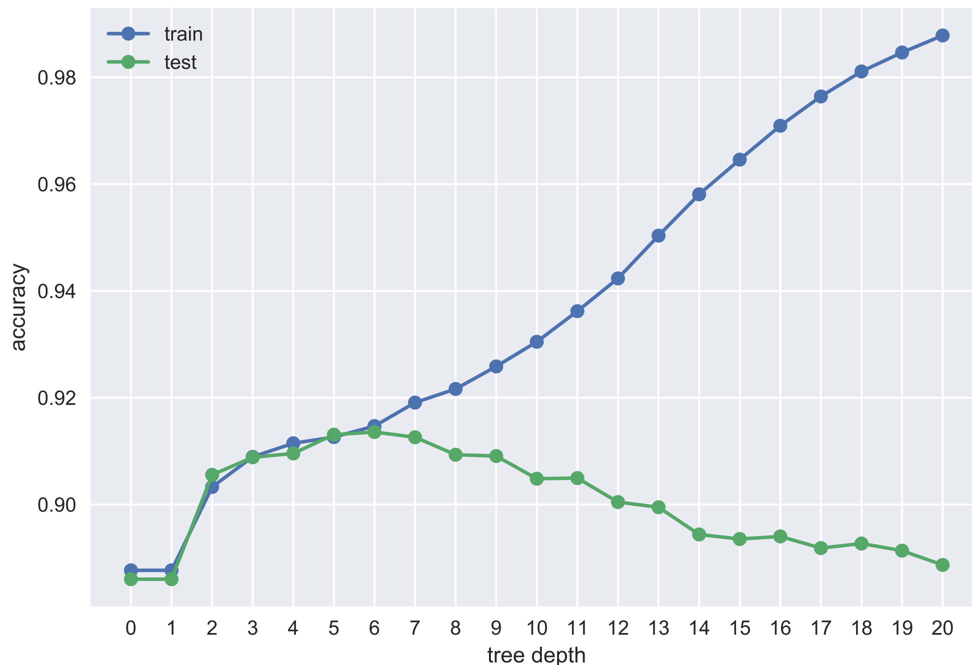


Figure 4: Training and test set accuracies for a decision tree classifier, as a function of the maximally allowed tree depth.

3 Decision trees for regression

Our discussion until now concerned *classifiers*: predictors with a discrete output. Let's turn to *regression* instead, where our goal is to predict a continuous numerical output. Regression trees have the same structure as classification tree. The only difference is that the leaf nodes now return a numerical output. Figure 5 shows a simple decision tree for regression: if the input variable x is greater than the threshold 0.3434, we return 11, otherwise we return 6.6.

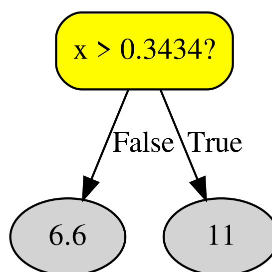


Figure 5: Example of a small decision tree for a regression task.

The training algorithm is almost the same for regression as for classification. The necessary modifications are the following:

- For one of the base cases in the recursion, we now check whether the variance[2] in the output Y is smaller than some threshold ε, instead of checking whether all outputs are identical.

[2]The sample variance of a sample $Y = y_1, \ldots, y_n$ is $V(Y) = \frac{1}{n}(y_i - \bar{y})^2$, where $\bar{y}$ is the sample mean of Y. It can also be computed via the formula $V(Y) = \frac{1}{n}\sum_{i=1}^{n} y_i^2 - \frac{1}{n^2}\left(\sum_{i=1}^{n} y_i\right)^2$.

- Instead of using the most frequent value when leaf nodes in the recursion base cases, we now use the *mean value* of the output Y.
- Instead of the homogeneity criteria we defined for classifiers, which all assumed discrete outputs, we now have to define criteria that work for continuous outputs.

The most commonly used criterion for regression trees, originally introduced in the CART algorithm (Breiman et al., 1984), is called *variance reduction*. It it uses the variance of the outputs in the same way that we used the entropy and Gini impurity previously: again, the idea is to try to find subsets that are homogeneous, meaning here that the variance is low. So the formula we use is

$$\text{variance-reduction}(Y, F) = V(Y) - \sum_{i=1}^{m} \frac{|Y_i|}{|Y|} V(Y_i)$$

Note the similarity to the information gain formula that we saw previously.

3.1 Plotting the output of a decision tree regressor

To illustrate the workings of a decision tree regression model, we can plot its output if the input is univariate or bivariate. The functions defined by decision tree regression models are *piecewise constant*: the output is constant in intervals, and then when we pass some threshold, we'll switch abruptly to a new output.

Figure 6 shows an example of the output of three different tree regressors for some univariate prediction problem. The training data is also included in the plot. As you can see, there seem to be some general tendencies but also a bit of noise. When the tree is complex (that is, it is deep), the model is able to memorize all the noise and we might run into problems with overfitting. Conversely, the first of the three trees probably is probably too simple (underfitting). In practice, we'll use cross-validation or a validation set to tune the complexity to get a good performance (e.g. mean squared error) on held-out data.

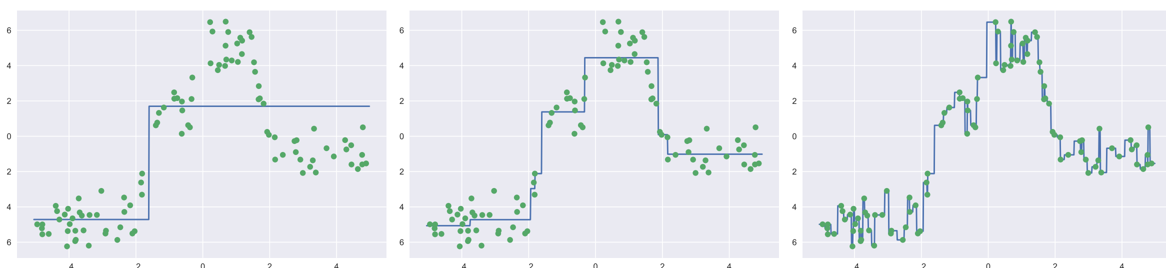


Figure 6: Example of underfitting, a good fit, and overfitting for decision tree regression models.

References

[Breiman et al.1984] Leo Breiman, J. H. Friedman, Olshen, R. A., and C. J. Stone. 1984. *Classification and regression trees*. Wadsworth & Brooks/Cole Advanced Books & Software.

[Quinlan1986] J. Ross Quinlan. 1986. Induction of decision trees. *Machine Learning*, 1(1):81–106.

[Quinlan1993] J. Ross Quinlan. 1993. *C4.5: Programs for Machine Learning*. Morgan Kaufmann.