Data Analysis and Machine Learning: Trees, 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 9, 2018

Decision trees, overarching aims

Decision trees are supervised learning algorithms used for both, classification and regression tasks where we will concentrate on classification in this first part of our decision tree tutorial. Decision trees are assigned to the information based learning algorithms which use different measures of information gain for learning. We can use decision trees for issues where we have continuous but also categorical input and target features.

Nodes, leafs, roots and branches

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 sub datasets are as pure as possible.

The descriptive feature which leaves the target feature most purely is said to be the most informative one. This 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**.

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 kind of 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 decision tree mainly contains of a **root node**, **interior nodes**, and **leaf nodes** which are then connected by **branches**.

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

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.preprocessing import PolynomialFeatures
from sklearn.linear_model import LinearRegression

steps=250

distance=0
x=0
distance_list=[]
steps_list=[]
while x<steps:
    distance+=np.random.randint(-1,2)
    distance_list.append(distance)
    x+=1
    steps_list.append(x)
plt.plot(steps_list,distance_list, color='green', label="Random Walk Data")

steps_list=np.asarray(steps_list)
distance_list=np.asarray(distance_list)

X=steps_list[:,np.newaxis]

#Polynomial fits

#Degree 2
poly_features=PolynomialFeatures(degree=2, include_bias=False)
X_poly=poly_features.fit_transform(X)

lin_reg=LinearRegression()
poly_fit=lin_reg.fit(X_poly,distance_list)
b=lin_reg.coef_
c=lin_reg.intercept_
print ("2nd degree coefficients:")
print ("zero power: ",c)
```

```
print ("first power: ", b[0])
print ("second power: ",b[1])

z = np.arange(0, steps, .01)
z_mod=b[1]*z**2+b[0]*z+c

fit_mod=b[1]*X**2+b[0]*X+c
plt.plot(z, z_mod, color='r', label="2nd Degree Fit")
plt.title("Polynomial Regression")

plt.xlabel("Steps")
plt.ylabel("Distance")

#Degree 10
poly_features10=PolynomialFeatures(degree=10, include_bias=False)
X_poly10=poly_features10.fit_transform(X)

poly_fit10=lin_reg.fit(X_poly10,distance_list)

y_plot=poly_fit10.predict(X_poly10)
plt.plot(X, y_plot, color='black', label="10th Degree Fit")

plt.legend()
plt.show()


#Decision Tree Regression
from sklearn.tree import DecisionTreeRegressor
regr_1=DecisionTreeRegressor(max_depth=2)
regr_2=DecisionTreeRegressor(max_depth=5)
regr_3=DecisionTreeRegressor(max_depth=7)
regr_1.fit(X, distance_list)
regr_2.fit(X, distance_list)
regr_3.fit(X, distance_list)

X_test = np.arange(0.0, steps, 0.01)[:, np.newaxis]
y_1 = regr_1.predict(X_test)
y_2 = regr_2.predict(X_test)
y_3=regr_3.predict(X_test)

# Plot the results
plt.figure()
plt.scatter(X, distance_list, s=2.5, c="black", label="data")
plt.plot(X_test, y_1, color="red",
         label="max_depth=2", linewidth=2)
plt.plot(X_test, y_2, color="green", label="max_depth=5", linewidth=2)
plt.plot(X_test, y_3, color="m", label="max_depth=7", linewidth=2)

plt.xlabel("Data")
plt.ylabel("Darget")
plt.title("Decision Tree Regression")
plt.legend()
plt.show()
```

Maxwell-Boltzmann velocity distribution

```
# Program to test the Metropolis algorithm with one particle at given temp in
# one dimension
#!/usr/bin/env python
import numpy as np
```

```
import matplotlib.mlab as mlab
import matplotlib.pyplot as plt
import random
from math import sqrt, exp, log
from sklearn.preprocessing import PolynomialFeatures
from sklearn.linear_model import LinearRegression
# initialize the rng with a seed
random.seed()
# Hard coding of input parameters
MCcycles = 100000
Temperature = 2.0
beta = 1./Temperature
InitialVelocity = -2.0
CurrentVelocity = InitialVelocity
Energy = 0.5*InitialVelocity*InitialVelocity
VelocityRange = 10*sqrt(Temperature)
VelocityStep = 2*VelocityRange/10.
AverageEnergy = Energy
AverageEnergy2 = Energy*Energy
VelocityValues = np.zeros(MCcycles)
# The Monte Carlo sampling with Metropolis starts here
for i in range (1, MCcycles, 1):
    TrialVelocity = CurrentVelocity + (2.0*random.random() - 1.0)*VelocityStep
    EnergyChange = 0.5*(TrialVelocity*TrialVelocity -CurrentVelocity*CurrentVelocity);
    if random.random() <= exp(-beta*EnergyChange):
        CurrentVelocity = TrialVelocity
        Energy += EnergyChange
        VelocityValues[i] = CurrentVelocity
    AverageEnergy += Energy
    AverageEnergy2 += Energy*Energy
#Final averages
AverageEnergy = AverageEnergy/MCcycles
AverageEnergy2 = AverageEnergy2/MCcycles
Variance = AverageEnergy2 - AverageEnergy*AverageEnergy
print(AverageEnergy, Variance)
n, bins, patches = plt.hist(VelocityValues, 400, facecolor='green')

plt.xlabel('$v$')
plt.ylabel('Velocity distribution P(v)')
plt.title(r'Velocity histogram at $k_BT=2$')
plt.axis([-5, 5, 0, 600])
plt.grid(True)
from collections import Counter

#print (Counter(VelocityValues))

print (VelocityValues[:20])
VelocityValues=list(Counter(VelocityValues).keys())
d=list(Counter(VelocityValues).values())

VelocityValues=np.asarray(VelocityValues)[:, np.newaxis]
d=np.asarray(d)
print (VelocityValues.shape, d.shape)

plt.scatter(VelocityValues, d)
plt.show()

#2nd Degree Polynomial
poly_feat=PolynomialFeatures(degree=20, include_bias=False)
X_poly=poly_feat.fit_transform(VelocityValues)
lin_reg=LinearRegression()
```

```
poly_fit=lin_reg.fit(X_poly,d)

y_plot=poly_fit.predict(X_poly)
plt.title("Polynomial Fit")
plt.plot(VelocityValues, y_plot, color='black', label="Fit")
plt.show()

#Decision Trees

from sklearn.tree import DecisionTreeRegressor
regr_1=DecisionTreeRegressor(max_depth=2)
regr_2=DecisionTreeRegressor(max_depth=5)
regr_3=DecisionTreeRegressor(max_depth=7)
regr_1.fit(VelocityValues, d)
regr_2.fit(VelocityValues, d)
regr_3.fit(VelocityValues, d)

X_test = np.arange(0.0, MCcycles, 0.01)[:, np.newaxis]
y_1=regr_1.predict(X_test)
y_2=regr_2.predict(X_test)
y_3=regr_3.predict(X_test)

plt.title("Decision Tree")
plt.plot(X_test, y_1, color="red", label="max_depth=2", linewidth=2)
plt.plot(X_test, y_2, color="green", label="max_depth=5", linewidth=2)
plt.plot(X_test, y_3, color="m", label="max_depth=7", linewidth=2)
plt.show()

#Separate each frequency not in one specific velocity, but in a range of values,
#i.e. frequency of all velocities in range -5 to -4.9, -4.9 to -4.8, etc...
```

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, $R_1, R_2, \ldots, R_J$.

1. 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 predic- tive 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 the jth box.

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 RSS 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.

A schematic 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 *alpha*, 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.

The zoo data

```
import pandas as pd
import numpy as np
from pprint import pprint
from sklearn.tree import DecisionTreeClassifier

#Import the dataset
dataset = pd.read_csv('data/zoo.csv')
#We drop the animal names since this is not a good feature to split the data on
#dataset=dataset.drop('animal_name',axis=1)
#Split the data into a training and a testing set
train_features = dataset.iloc[:80,:-1]
test_features = dataset.iloc[80:,:-1]
train_targets = dataset.iloc[:80,-1]
test_targets = dataset.iloc[80:,-1]
```

```
#Train the model
tree = DecisionTreeClassifier(criterion = 'entropy').fit(train_features,train_targets)
#Predict the classes of new, unseen data
prediction = tree.predict(test_features)
#Check the accuracy
print("The prediction accuracy is: ",tree.score(test_features,test_targets)*100,"%")
```

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.

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.

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.

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.

A simple scikit-learn example

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.preprocessing import LabelEncoder
from sklearn.model_selection import cross_validate
# Data set not specificied
X = dataset.XXX
Y = dataset.YYY
#Instantiate the model with 100 trees and entropy as splitting criteria
Random_Forest_model = RandomForestClassifier(n_estimators=100,criterion="entropy")
#Cross validation
accuracy = cross_validate(Random_Forest_model,X,Y,cv=10)['test_score']
```

Boosting

More material to come here.