Chapter 10: Statistics for Tree-based

Methods

In the previous section, we covered some important concepts in classification models. We also built a naïve Bayes’ classifier from scratch. This kind of practice is very important, because it requires you to understand every aspect of the details.

In this section, we are going to dive into another family of statistical models which are also widely used in statistical analysis as well as machine learning: the tree-based models. The tree-based models can be used for both classification tasks and regression tasks.

In this chapter, you are going to:

1. Gain an overview of tree-based classification
2. Understand the details of classification tree building
3. Understand the mechanism of regression tree
4. Know how to use scikit-learn library to build and regularize a tree-based method

Let’s get started!

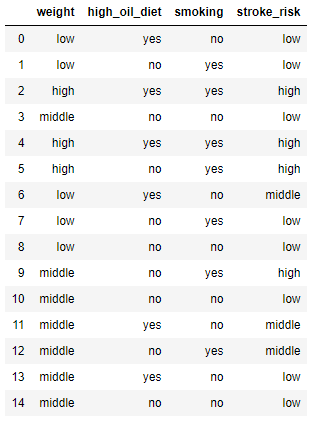
Overview of Tree-based Methods for Classification Tasks

Tree-based methods have two major variates, classification tree and regression tree. A classification tree predicts categorical outcomes from a finite set of possibilities while a regression tree predicts numerical outcomes. Let’s first look at the classification tree, especially its quality that makes it superior to other classification methods like simple logistic regression classifier and naïve Bayes’ classifier.

A classification tree creates a set of rules and partitions the data into various subspace in the feature space (or feature domain) in an optimal way.

First question, what is a feature space?

Let’s take our stroke risk data we used in last chapter as sample data. I pasted the dataset again displayed below.

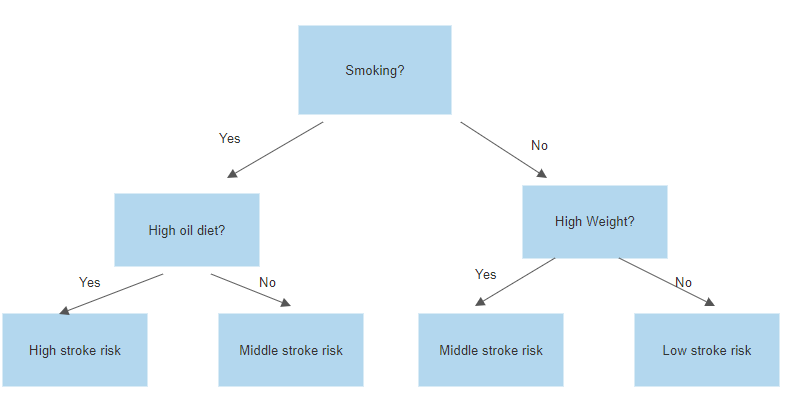


Stroke risk data

We have 3 features. If we only look at the weight, it can take three different levels: low, middle and high. Imagine in a one-dimensional line representing weight, there are only three discrete points the value can take, namely the three levels. This is a one-dimensional feature space or feature domain.

On another hand, high-oil dieting habit and smoking habit are another two-feature dimension with two possibilities to take, respectively. Therefore, a person can be on one of 12 (3\*2\*2) combinations of all features in this three-dimensional feature space.

A classification tree is built with rules to map these 12 points in the feature space to the outcome space which has 3 possible outcomes. Each rule is a yes-no question and the answer will be non-ambiguous, so each data record has a certain path to do down the tree. Below is an example of such classification tree.



An example of classification tree for stroke risk data

Now, it is a good time to introduce some terminologies that are listed below. A tree is usually drawn upside down, but it is a good thing you follow down a chain of decisions to reach the final status.

1. Root node. A root node is the only node/block that only has outgoing arrows. In the tree above, it is the one at top with text Smoking. The root node contains all the records and they haven’t been divided into sub-categories, which corresponds to partitions of feature space.
2. Decision node. A decision node is one node with both incoming arrow and outgoing arrows. It splits the data feed into two groups. For example, the two nodes on the second level of the tree are decision nodes. The one on the left splits the smoking group further into the smoking & high-oil diet group and the smoking but non-high oil diet group. The one on the right splits the non-smoking group further into the non-smoking & high weight and non-smoking & non-high weight groups.
3. Leaf Node. A leaf node or a leaf is a node with only incoming arrows. A leaf node represents the final terminal of a classification process where no further splitting is needed or allowed. For example, the node at the bottom left is a leaf which indicates that people who smoke and have high oil diet are classified to have high risk of stroke. It is not necessary for a leaf to only contain pure results. In this case, it is OK to have low stroke risk and high stroke risk people in the leaf. What we optimized is the pureness of the classes in the node. The label for the records in a leaf node is the majority label. If there is a tie, a common solution is to pick a random one of the tied candidates.
4. Parent Node and Children Node. A node at the start of arrows is the parent node of the nodes at the end of arrows, which are called children node. A node can be simultaneously a parent node and a child node, except the root node and the leaf. The process of determining which feature/criteria to use to generate children nodes is called splitting. It is common practice to do binary splitting, which means a parent node will have two child nodes.
5. Depth and Pruning. The depth of a decision tree is defined as the length of the chain from the root node to the furthest leaf. In the stroke risk case, the depth is 2. It is not necessary for a decision tree to be balanced. One branch of the tree can have a bigger depth of another branch if accuracy requires. The operation of removing children nodes, including grandchildren nodes and more, is called pruning, just like pruning a biological tree.

One good thing about a decision tree is its universality. The features don’t necessarily take discrete values, they can also take continuous numerical values. For example, if the weight is replaced with continuous numerical values, the splitting on high weight or not will be replaced by a node with criteria like weight > 200 pounds?

Now, let’s go over the advantages of decision trees.

1. The biggest advantage of decision trees is they are easy to understand. For a person without any statistics or machine learning background, decision trees are the easiest classification algorithms to understand.
2. The decision tree is not sensitive to data preprocessing and data incompletion. For many machine learning algorithms, data preprocessing is vital. For example, the unit of feature in gram or kilogram will influence the coefficient values of logistic regression. However, decision trees are not sensitive to the data preprocessing. The selection of the criteria will adjust automatically when the scale of the original data changes, but the splitting results will be unchanged. If we apply logistic regression on the stroke risk data, a missing value of a feature will break the algorithm. However, decision trees are more robust to achieve relatively stable result. For example, if a person who doesn’t smoke misses the weight data, he or she can be classified into the low risk or middle risk groups randomly ( of course there are better ways to decide like select the mode of records similar to it), but he or she won’t be classified into high risk groups. This result is sometimes good enough for practical use.
3. Explainability. When a decision tree is trained, you are not only getting a model, you also get a set of rules which you can explain to your boss or supervisor. This is also why I love the decision tree the most: explainability. The importance of features can also be extracted. For example, in general, the closer the feature is to the root, the more important the feature is in the model. In the stroke risk example, smoking is the root node which enjoys the highest feature importance. We will talk about how the positions of the features are decided in next chapter.

Now, let’s also talk about the disadvantages of the decision tree.

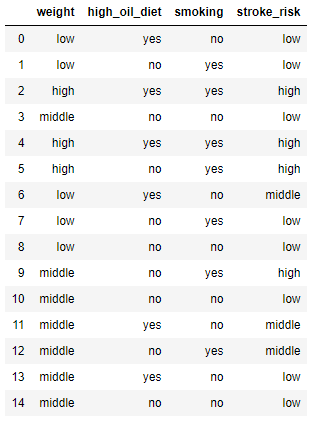
1. It is easy to overfit. Without control or penalization, decision trees can be very complex. How? Imagine that unless there are two records with exact the same features but different outcome variable, the decision tree can actually build one leaf node for every record to reach 100% accuracy on the training set! However, the model very likely will not be generalized to another data set. Pruning is a common approach to remove over-complex sub-branches. There are also constraints on the splitting step, which we will discuss soon.
2. Greedy approach not necessarily gives the best model. A single decision tree is built by greedily selecting the best splitting feature sequentially. As a combination problem with exponential number of possibilities, greedy approach doesn’t necessarily give the best model. In most cases, this isn’t a problem. In some cases, a small change in the training dataset might generate a completely different decision tree and give a different set of rules. Make sure you double check it before presenting it to your boss!

To understand why building a decision tree involves combinational number of choices. Let’s build a decision tree with a depth of 3, trained on a dataset of 3 continuous variable features. We have 3 decision nodes including the root node to generate 4 leaves. Each decision node can choose from 3 features for splitting. Therefore, resulting in a total of 27 possibilities. Yes, one children node can choose the same feature as its parent.  
  
Imagine we have 4 features, the total number of choices become 64. If the depth of the tree increases by 1, then we add 4 more decision nodes. Therefore, the total number of splitting feature choices is 16384, which is huge for such a 4-feature dataset. Most trees will be obviously useless, but greedy approach doesn’t guarantee the generation of the best decision tree.

We have covered the terminologies, advantages and disadvantages of decision trees. In next chapter, we will go deep into the decision tree, specifically how splitting is decided.

Details of a Classification Tree

Let’s start by examining the dataset one more time.



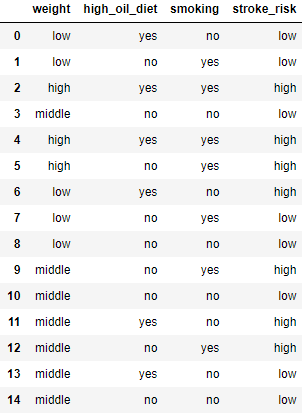
Stroke risk data

For the purpose of this demonstration, I will just group the middle-risk and high-risk patients to the high-risk group. This way, the classification problem becomes a binary classification problem, which is easier to explain. After going through this part, you can do the exercises on the original 3-category problem as an exercise.

The following code snippet generates the new dataset.

df["stroke\_risk"] = df["stroke\_risk"].apply(lambda x: "low" if x == "low" else "high")

The new dataset looks as below.



Binary stroke risk data

Now, let’s think about the root node. Which feature and what kind of criteria should we choose to generate two children nodes?

How splitting works

The principle of splitting is that splitting as a feature must get us closer to full correct classification. We need a numerical metric to compare different choices of splitting features. The goal for classification is to classify records to pure states such that each leaf will contain records as pure as possible. Therefore, pureness or impureness becomes a natural choice of metric.

The most common metric is called Gini impurity. It measures how impure a set of data is. For a binary-class set of data with class label A and B, the definition of Gini impurity is the following

If the set only contains A or B, the Gini impurity is 0. The maximal impurity is 0.5 when half of the records are A and another half are B. For a 3-class dataset, the minimum is still 0 but the maximum becomes .

Gini impurity is named after the Italian demographer and statistician, Corrado Gini. Another more well-known index named after him is the Gini index, which measures the inequality of wealth distribution in a society.

Let’s see how this unfolds. At the root node, without any splitting. The Gini impurity is calculated as because we have 8 low-risk records and 7 high-risk records, the value is about 0.498, close to the highest possible impurity.

After splitting by one criterion, we have two children nodes. The way to obtain the new, lower impurity is to calculate the weighted Gini impurity of the two children nodes.

First, let’s take the high oil diet for example, Let’s examine the partition of the high oil diet group. The following code snippet does the counting.

Counter(df[df["high\_oil\_diet"]=="yes"]["stroke\_risk"])

There is a total of 6 records with 2 low-risk records and 4 high-risk records. Therefore, the impurity for the high oil diet group is .

Meanwhile, we can calculate the non-high oil diet group’s statistics. Let’s select and count them using the following code snippet.

Counter(df[df["high\_oil\_diet"]=="no"]["stroke\_risk"])

There is a total of 9 records with 6 low-risk records and 3 high-risk records. Note that the proportionalities are the same for the high oil diet but with exchanging groups. Therefore, the Gini impurity is also .

The weighted Gini impurity remains because . It is about 0.444.

So what do we get from such a classification? We have reduced the Gini impurity from 0.498 to 0.444, which is just a slight decrease, better than nothing.

Next, let’s examine the smoking behavior.

By the same token, let’s first check the smoking group’s statistics. The following code snippet does the counting.

Counter(df[df["smoking"]=="yes"]["stroke\_risk"])

There is a total of 7 smoking cases. 5 of them are of high stroke risk and 2 of them are of low stroke risk. The Gini impurity is therefore about 0.408.

Let’s check the non-smokers. Counter(df[df["smoking"]=="no"]["stroke\_risk"])

There are 8 non-smokers and 6 of them are of low stroke risk and 2 of them are of high stroke risk, therefore the Gini impurity is 0.375.

Let’s obtain the weighted impurity, it is about . This is a 0.108 decrease from the original impurity without splitting and it is better than the splitting on high oil diet.

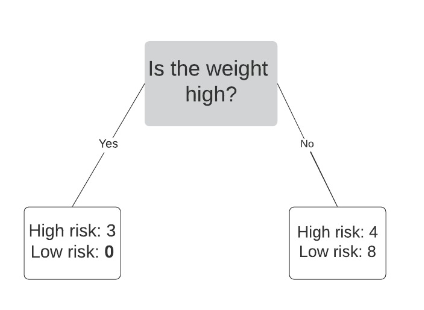
I will omit the calculation for the other feature weight, but I will list the result for you in the following table. Note that the feature weight has 3 levels so there can be multiple rules for splitting the feature. Here I list all of them.

In the Yes and No group statistics, I list the number of high-stroke risk records, number of low-risk records and Gini impurity value separated by commas.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Splitting feature | Splitting rule | The Yes group | The No group | The weighted total Gini impurity |
| Smoking | Is Smoking? | 5, 2, 0.408 | 2, 6, 0.375 | 0.390 |
| High oil diet | Is high oil diet? | 4, 2, 0.444 | 3, 6, 0.444 | 0.444 |
| Weight | Is low weight? | 1, 4, 0.320 | 6, 4, 0.480 | 0.427 |
| Weight | Is middle weight? | 3, 4, 0.490 | 4, 4, 0.50 | 0.495 |
| Weight | Is high weight? | 3, 0, 0 | 4, 8, 0.444 | 0.356 |

The Gini impurity evaluation table for different splitting features at root node

Note that I highlighted the Gini impurity for the high weight group and the weighted Gini impurity for the last splitting choice. All high weight patients have a high stroke risk, and this drives the weighted impurity down to 0.356, the lowest of all possible splitting rules. Therefore, we choose the last rule to build our decision tree. After the first splitting, the decision tree now looks like the following.

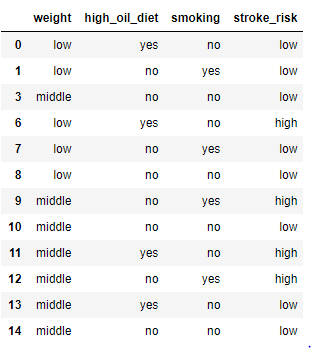


Decision Tree after the first splitting

Note that the left branch now contains a pure node, which becomes a leaf. Therefore, our next stop only focuses on the right branch. We naturally have an imbalanced tree now.

Now, we have 4 choices for the splitting of 12 records. First, I will select these 12 records out with the one-line code snippet. df\_right = df[df["weight"]!="high"]

The result looks as following. The Gini impurity for the right splitting node is 0.444 as calculated above. This will become our new baseline.



Low-weight and middle-weight group

Like what we did earlier, let’s build a table to compare different splitting choices for the splitting node on the right. The ordering of the numbers is the same as the previous table.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Splitting feature | Splitting rule | The Yes group | The No group | The weighted total Gini impurity |
| Smoking | Is Smoking? | 2, 2, 0.5 | 2, 6, 0.375 | 0.417 |
| High oil diet | Is high oil diet? | 2, 2, 0.444 | 2, 6, 0.375 | 0.417 |
| Weight | Is low weight? | 1, 4, 0.320 | 3, 4, 0.490 | 0.420 |
| Weight | Is middle weight? | 4, 4, 0.490 | 1, 4, 0.320 | 0.420 |

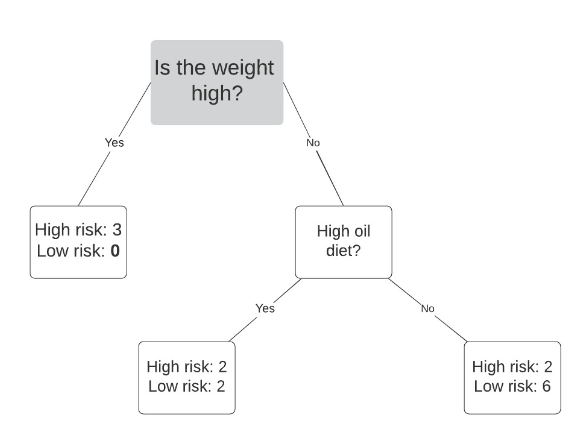
The Gini impurity evaluation table for different splitting features at right splitting node

We essentially only have 3 choices because the two splitting rules on the feature weight are mirrors of each other.

Now, we have a tie. We can randomly select one criterion for building the trees further. This is one reason that decision trees don’t theoretically generate the best results.

However, an intuitive way to solve this issue is to build the both possibilities and even more possible trees which violates greedy approach and let them vote on the prediction results. This is a common method to build a more stable model, or an ensemble of models. We will cover related techniques in next chapter.

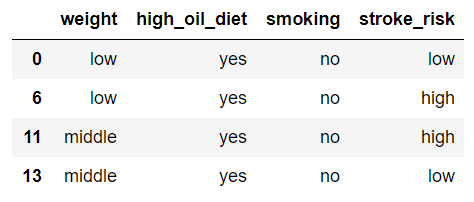
Let’s say I choose high oil diet as the criteria. The tree now looks like the following.



Decision tree after the second splitting

Now, let’s look at the two newly generated nodes. The first one at a depth of 2 contains 2 high stroke risk records and 2 low stroke risk records. They don’t have heavy weight, but a high oil diet. Let’s check out their profile with this line of code df\_right[df\_right["high\_oil\_diet"]=="yes"]

The result looks like the following:

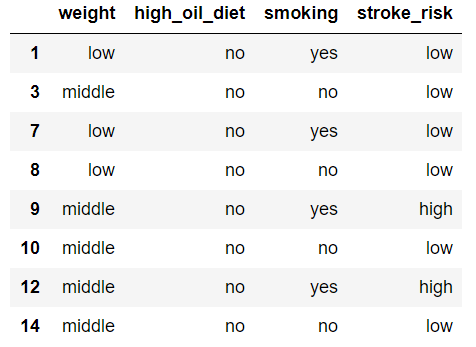


Records classified into the first node at depth of 2

Note that the low-weight category contains one low stroke risk record and a high stroke risk example. The same situation happens to the middle-weight category. This makes the decision tree incapable of further splitting on any feature. There won’t be any Gini impurity decreasing for splitting. Therefore, we can stop here for this node.

Well, what if we want to continue improving the classification results. As you just discovered, there is no way that decision tree can classify these four records and no other machine learning can do it either. The problem is in the data, not in the model. There are two main approaches to solve this issue. The first option is to try to obtain more data. With more data, we may likely find that low weight is positively correlated with low stroke risk and further splitting on the feature weight might benefit decreasing the Gini impurity. Obtaining more data is always better because your training model got to see more data therefore reduces possible bias. Another option is to introduce more features. This essentially expand the feature space by more dimensions. For example, blood pressure might be another useful feature that might help us further increase the accuracy of the decision tree.

Now, let’s look at the second node at depth 2. The records classified into this node are the following, given by the code df\_right[df\_right["high\_oil\_diet"]!="yes"]



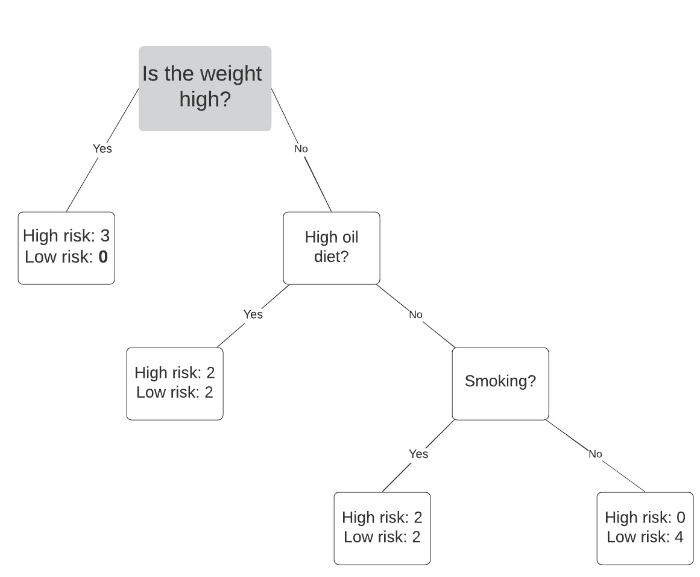
Records classified into the second node at depth of 2

Note that only 2 high stroke risk records are in this node. If we stop here, the Gini impurity is , which is quite a low value.

If we further split on the smoking feature, note that the all non-smokers, 4 of them, have a low stroke risk. Half of the smokers have high stroke risk and another half have low stroke risk. This will give us a weighted Gini impurity 0.25 if splitting on smoking.

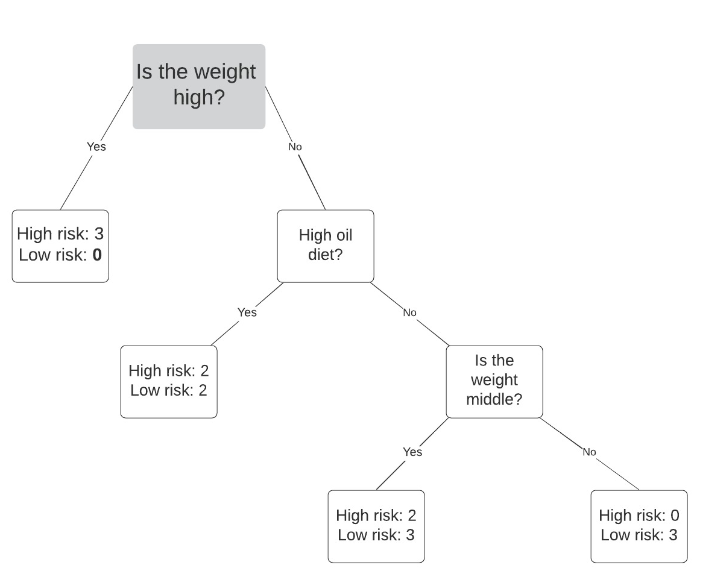
If we further split on the weight feature, all the low-weight patients are at low stroke risk. 2 out of 5 middle-weight records are at high risk. This will give us a weighted Gini impurity of , which is also not bad.

For the two cases, the final decision trees look as following. The immediate following decision tree has smoking as the last splitting feature.



Final decision tree, version 1

The next one is splitting on weight again at the second node at depth 2.



Final decision tree, version 1

Now, we need to make some hard choices to decide the final shape of our decision trees.

Evaluation of Decision Tree Performance.

In this section, let’s evaluate the performance of the decision tree classifiers. If we stop at depth 2. We have the following confusion matrix. Note that for the unclassifiable first node at depth 2, we can randomly assign it a label. Here. I assign it as high stroke risk.

|  |  |  |
| --- | --- | --- |
|  | Ground truth high risk | Ground truth low risk |
| Predicted high risk | 5 | 2 |
| Predicted low risk | 2 | 6 |

Confusion matrix of decision tree of depth 2

Generally, we identify the high risk as positive so the precision, recall, and f1 score are all . If you forget these concepts, you can review previous chapters.

If we don’t stop at depth of 2, the two finer decisions trees will have the following confusion matrices. Again, we assign the unclassifiable first node at depth 2, the label high stroke risk. However, the first node at depth 3 is also unclassifiable, because it contains equal high stroke risk and low stroke risk records. If they are classified as low risk ones, then we essentially obtain the same result as the depth of 2 one. Therefore, we assign the first leaf node at depth 3 a high stroke risk value.

|  |  |  |
| --- | --- | --- |
|  | Ground truth high risk | Ground truth low risk |
| Predicted high risk | 7 | 5 |
| Predicted low risk | 0 | 3 |

Confusion matrix of decision tree of depth 3, version 1

Note that we will have perfect recall, but the precision will be just slightly better than random guess, . The F1 score is .

Next, let’s check our final version. If we split with weight, the corresponding confusion matrix looks like the table below.

|  |  |  |
| --- | --- | --- |
|  | Ground truth high risk | Ground truth low risk |
| Predicted high risk | 5 | 2 |
| Predicted low risk | 2 | 6 |

Confusion matrix of decision tree of depth 3, version 1

The precision recall and F1 score will be identical to the depth 2 decision tree. In real life, we usually prefer the simplest model possible if it is as good or almost as good as the complicated ones. Although the first depth 3 decision tree has a better F1 score, but it also introduces one more unclassifiable node and one more rule. The second depth 3 decision tree does no better than the depth 2 one.

To constrain the complexity of the decision tree, there are usually three methods:

1. Constrain the depth of the tree. This is probably the most direct way of constraining the complexity of the decision tree.
2. Constrain the lower bound of the number of records classified into a node. For example, if after splitting, one child node will only contain very few data points, then it is likely not a good splitting.
3. Constrain the lower bound of information gain. In our case, the information gain means lower Gini impurity. For example, if we set a criterion that each splitting must lower the information gain by 0.1 then the splitting will likely stop soon, therefore confine the depth of the decision tree.

We will see algorithmic examples on more complex dataset later in this chapter.

When the number records in a splitting node is small, the Gini impurity reduction is no longer very representative. It is the same idea as in statistical significance. The larger the sample size is, the more confident we are about the derived statistics.  
  
You may also hear the size of the decision tree. Usually the size is not the same as the depth. The size refers to the total number of nodes in a decision tree. For a symmetric decision tree, the relationship is exponential.

Regression Tree

The regression tree is very similar to a classification tree. A regression tree takes numerical features as input and predicts another numerical variable.

It is perfectly fine to have mix-type features like some of them are discrete and some of them are continuous. We won’t cover such examples due to space limitation, but the intuition is straight forward. I will leave one example in the exercising chapter.

There are two very important visible differences:

1. The output is not discrete labels but numerical values
2. The splitting rules are “not” similar Yes or No questions. They are usually inequalities for values of certain features.

In this section, we will just use a one-feature dataset to build a regression tree that logistic regression classifier won’t be able to classify. I created an artificial dataset with the following code snippet.

def price\_2\_revenue(price):

if price < 85:

return 70 \* abs(price - 75)

elif price < 95:

return 10 \* 80

else:

return 80 \* (105 - price)

prices = np.linspace(80,100,8)

revenue = np.array([price\_2\_revenue(price) for price in prices])

plt.rcParams.update({'font.size': 22})

plt.figure(figsize=(10,8))

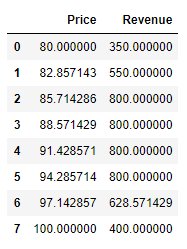
plt.scatter(prices,revenue,s=300)

plt.xlabel("price")

plt.ylabel("total revenue")

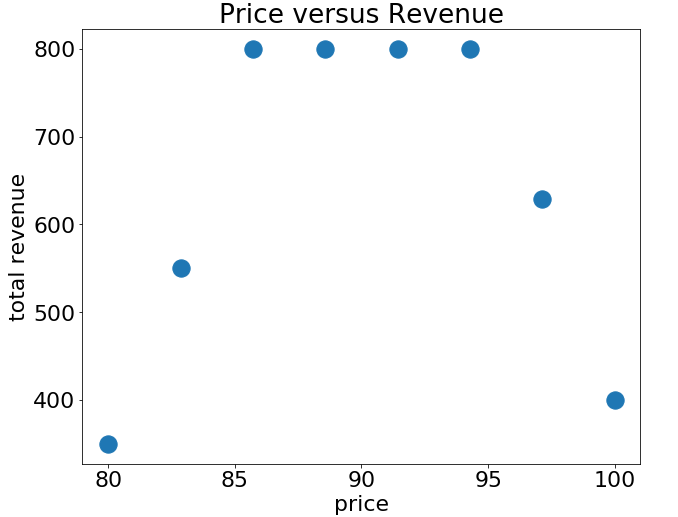
plt.title("Price versus Revenue");

Let’s say we want to investigate the relationship between the price of a good and its total revenue a day. If the price is set too low, the revenue will be lower because the price is low. If the price is too high, the revenue will also become small due to smaller amounts of the good being sold. The DataFrame looks as below.



Price and Total Revenue DataFrame

The following visualization makes such scenario clearer.



Relationship between Price and Revenue

The relationship between price and revenue is clearly non-linear and logistic regression won’t be able to classify it. A linear regression will likely become a horizontal line. There are clearly three regions that might differ relationships between revenue and price apply.

Now, let’s build a regression tree. Like the deduction of Gini impurity in classification tree, we need a metrics to measure the benefit of splitting. A natural choice is still the sum of squared residuals.

Let’s start from the root node. We have 8 data points, so there are essentially 7 intervals where we can put the first splitting criteria into. For example, we can split at price = 85. Then we use the average revenue on both sides to be our prediction like the following. The code snippet for the visualization reads as the following.

plt.rcParams.update({'font.size': 22})

plt.figure(figsize=(12,8))

plt.scatter(prices,revenue,s=300)

plt.xlabel("price")

plt.ylabel("total revenue")

plt.title("Price versus Revenue")

threshold = 85

num\_left = sum(prices < threshold)

ave\_left = np.mean(revenue[prices < threshold])

num\_right = sum(prices > threshold)

ave\_right = np.mean(revenue[prices > threshold])

plt.axvline(threshold,color="red",linewidth=6)

plt.plot(prices[prices < threshold], [ave\_left for \_ in range(num\_left)],

linewidth=6,linestyle=":",c="orange",

label= "average revenue on the left half")

plt.plot(prices[prices > threshold], [ave\_right for \_ in range(num\_right)],

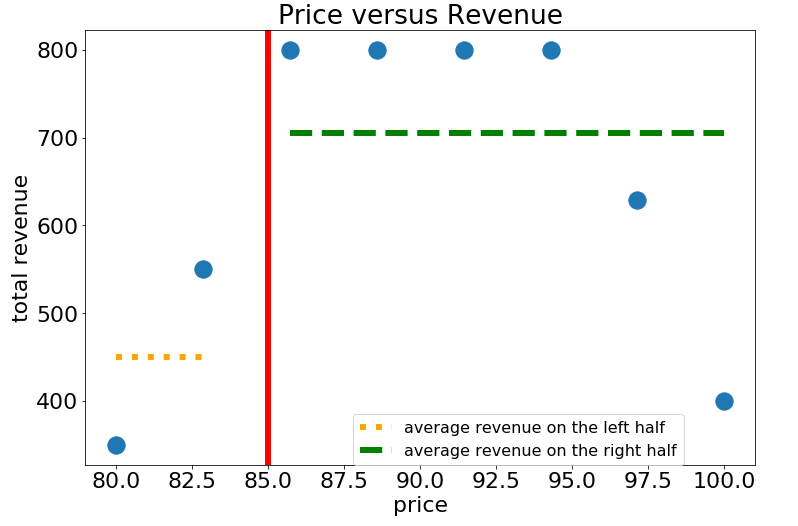
linewidth=6,linestyle="--",c="green",

label="average revenue on the right half");

plt.rcParams.update({'font.size': 16})

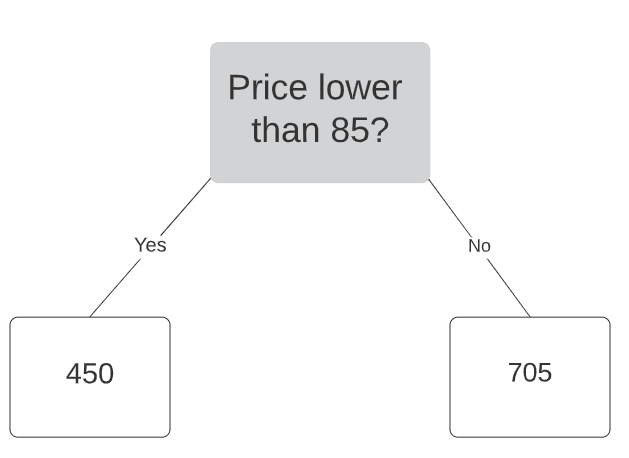
plt.legend(loc=[0.4,0]);

The dotted line represents the average price for the scenario when the price is smaller than 85.0. The dashed line represents the average price for the scenario when the price is larger than 85.0.



Splitting at price 85.0

If we stop here, the regression tree will have a depth of 1 and looks like the following.



Regression tree of depth 1

However, we haven’t tested the other 6 splitting choices. Any splitting choice will have a corresponding sum of squared residuals and we would like to go over all possibilities to determine THE splitting that gives the minimal sum of squared residuals.

Unlike Gini impurity where we need to take a weighted average, the total sum of squared residuals is a simple summation. Gini impurity is not additive because it only takes value between 0 and 1. Squared residuals are additive because each residual corresponds to one data point.

The following code snippet plots the sum of squared residuals against different choices of splitting. For completion, I plotted more than 7 splitting values to visualize the stepped pattern.

def cal\_ssr(arr):

if len(arr)==0:

return 0

ave = np.mean(arr)

return np.sum((arr-ave)\*\*2)

splitting\_values = np.linspace(80,100,20)

ssr\_values = []

for splitting\_value in splitting\_values:

ssr = cal\_ssr(revenue[prices < splitting\_value]) + cal\_ssr(revenue[prices > splitting\_value])

ssr\_values.append(ssr)

plt.rcParams.update({'font.size': 22})

plt.figure(figsize=(12,8))

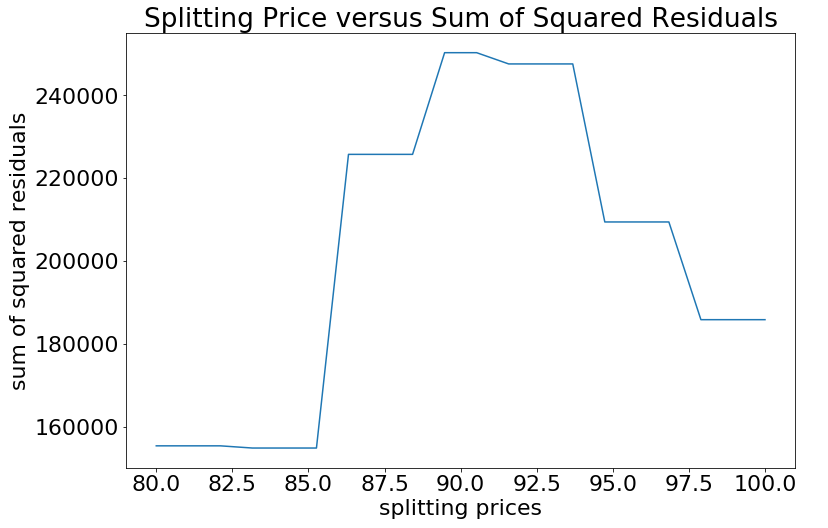
plt.xlabel("splitting prices")

plt.ylabel("sum of squared residuals")

plt.title("Splitting Price versus Sum of Squared Residuals")

plt.plot(splitting\_values,ssr\_values);

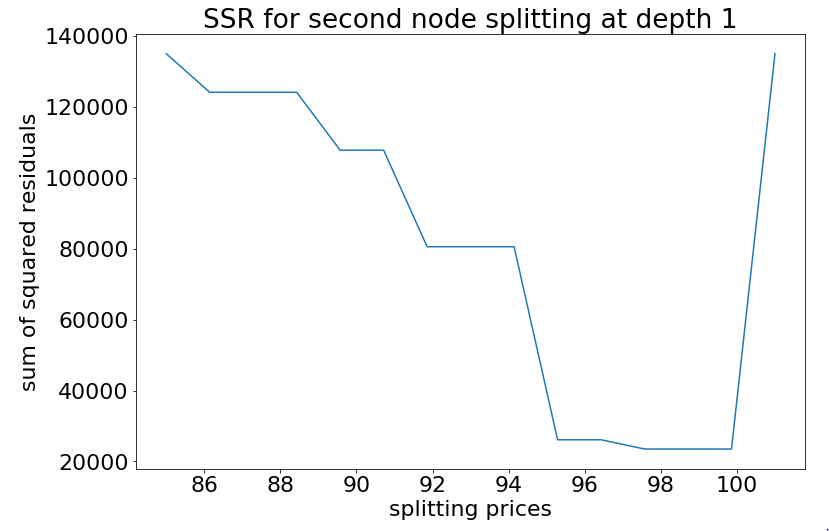
The result looks as the following.



Splitting Value for Root Node versus Sum of Squared Residuals

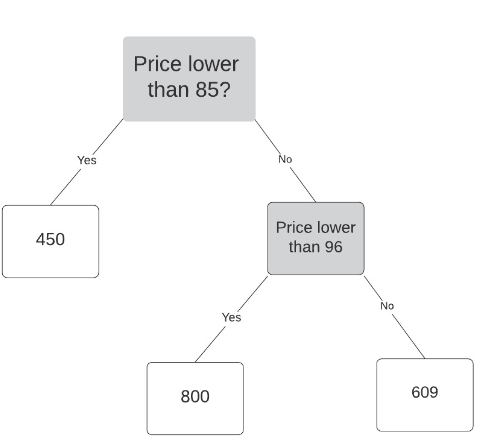
The above visualization indicates that 85.0, or any value between the second point and the third point, is the best splitting value for root node.

There are only two records in the first node at depth of 1, so we focus on the second node, and repeat the process above. The code is omitted due to space limitation. The visualization of the sum of squared residuals is the following.



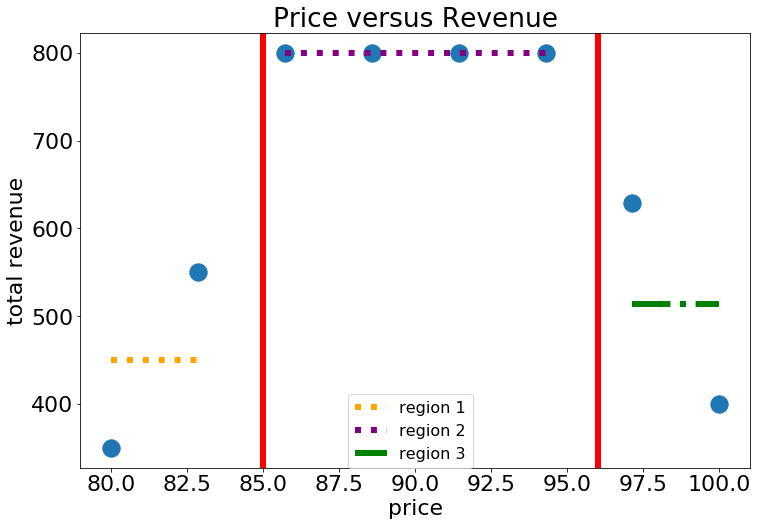
Splitting Choices at the Second Node at Depth 1

Now, in order to achieve the minimal sum of squared error, we should put the last data point into one children node. However, you see that if we split at 98, the penalty we pay is not increasing much. If we include another one like splitting at 96, the penalty will soar. It may be a good idea to split at 96 rather than 98 because a leaf node containing too few records is not representative in general and often indicates overfitting. Here is the final look of our regression tree. You can calculate the regressed average prices at each region easily.



Final Regression Tree

Below is a visualization for the partition of the regions.



Regressed Values and Region Partitioning

In multi-feature cases, we will have more than one feature. The scanning of the best splitting value should include all the features, but the idea is the same.

Examples with Scikit-learn

Before ending this chapter, let’s try some Scikit-learn APIs. You can verify that the results agree with our models built from scratch. The following code snippet builds a regression tree of maximal depth of 1 on the price-revenue data.

from sklearn.tree import DecisionTreeRegressor

from sklearn import tree

prices, revenue = prices.reshape(-1,1), revenue.reshape(-1,1)

regressor = DecisionTreeRegressor(random\_state=0,max\_depth=1)

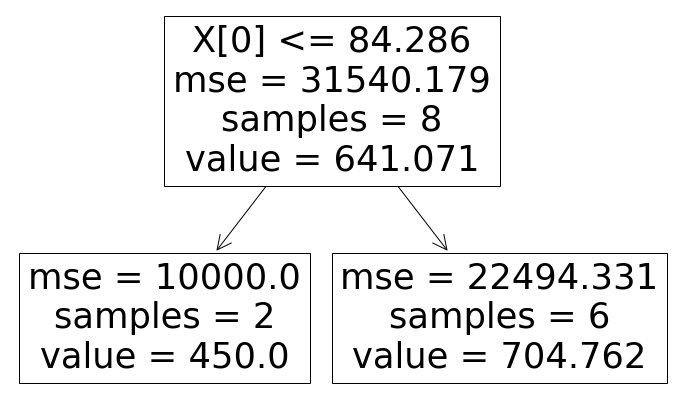
regressor.fit(prices,revenue)

Now, we can visualize the tree with the following code snippet.

plt.figure(figsize=(12,8))

tree.plot\_tree(regressor);

The tree structure looks as below.

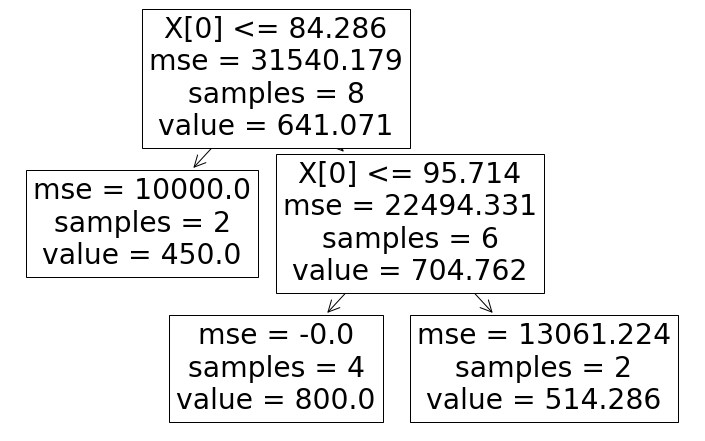


Regression Tree Visualization of Depth 1

Next, we limit the maximal depth to 2 and require the minimal number of records/samples in a leaf node to be 2. The code only requires a small change in the following line.

regressor = DecisionTreeRegressor(random\_state=0,max\_depth=2,min\_samples\_leaf=2)

We obtain the following tree structure.



Regression Tree Visualization of Depth 2

As you can see, this agrees exactly as the one we built from scratch.

Scikit-learn decision tree API can’t explicitly handle categorical variables. There are various options like One-Hot encoding to bypass this limitation. You are welcome to explore the solutions on your own.

Summary

In this chapter, we started with fundamental concepts in decision trees, built a simple classification tree and a regression tree from scratch. We went over the details and checked the consistency with the Scikit-learn library API.

You may notice that tree methods do tend to overfit and might fail to reach the optimal model. In next chapter, we will explore the so-called ensemble learning. They are meta-algorithms that can be used on top of many other machine learning algorithms as well.