**Decision Tree Classification in Python**

In this tutorial, learn Decision Tree Classification, attribute selection measures, and how to build and optimize Decision Tree Classifier using Python Scikit-learn package.

As a marketing manager, you want a set of customers who are most likely to purchase your product. This is how you can save your marketing budget by finding your audience. As a loan manager, you need to identify risky loan applications to achieve a lower loan default rate. This process of classifying customers into a group of potential and non-potential customers or safe or risky loan applications is known as a classification problem.

Classification is a two-step process, learning step and prediction step. In the learning step, the model is developed based on given training data. In the prediction step, the model is used to predict the response for given data. Decision Tree is one of the easiest and popular classification algorithms to understand and interpret. It can be utilized for both classification and regression kind of problem.

In this tutorial, you are going to cover the following topics:

Decision Tree Algorithm

How does the Decision Tree algorithm work?

Attribute Selection Measures

Information Gain

Gain Ratio

Gini index

Optimizing Decision Tree Performance

Classifier Building in Scikit-learn

Pros and Cons

Conclusion

**Decision Tree Algorithm**

A decision tree is a flowchart-like tree structure where an internal node represents feature(or attribute), the branch represents a decision rule, and each leaf node represents the outcome.

The topmost node in a decision tree is known as the root node. It learns to partition on the basis of the attribute value. It partitions the tree in recursively manner call recursive partitioning. This flowchart-like structure helps you in decision making. It's visualization like a flowchart diagram which easily mimics the human level thinking. That is why decision trees are easy to understand and interpret.

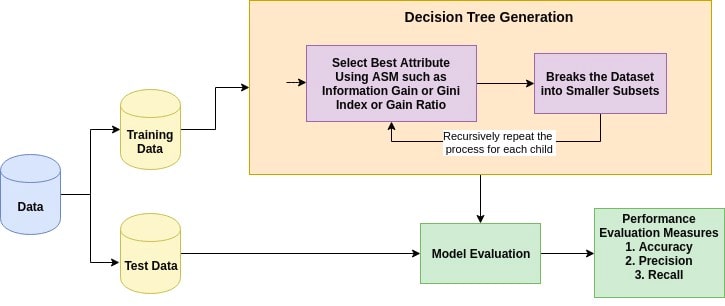


Decision Tree is a white box type of ML algorithm. It shares internal decision-making logic, which is not available in the black box type of algorithms such as Neural Network. Its training time is faster compared to the neural network algorithm. The time complexity of decision trees is a function of the number of records and number of attributes in the given data. Decision trees can handle high dimensional data with good accuracy.

**How does the Decision Tree algorithm work?**

The basic idea behind any decision tree algorithm is as follows:

1. Select the best attribute using Attribute Selection Measures(ASM) to split the records.
2. Make that attribute a decision node and breaks the dataset into smaller subsets.
3. Starts tree building by repeating this process recursively for each child until one of the condition will match:
   * + All the tuples belong to the same attribute value.
     + There are no more remaining attributes.
     + There are no more instances.



**Attribute Selection Measures**

Attribute selection measure is a heuristic for selecting the splitting criterion that partition data into the best possible manner. It is also known as splitting rules because it helps us to determine breakpoints for tuples on a given node. ASM provides a rank to each feature(or attribute) by explaining the given dataset. Best score attribute will be selected as a splitting attribute. In the case of a continuous-valued attribute, split points for branches also need to define. Most popular selection measures are Information Gain, Gain Ratio, and Gini Index.

**Information Gain**

Shannon invented the concept of entropy, which measures the impurity of the input set. In physics and mathematics, entropy referred as the randomness or the impurity in the system. In information theory, it refers to the impurity in a group of examples. Information gain is the decrease in entropy. Information gain computes the difference between entropy before split and average entropy after split of the dataset based on given attribute values.

The attribute A with the highest information gain, Gain(A), is chosen as the splitting attribute at node N().

**Gain Ratio**

Information gain is biased for the attribute with many outcomes. It means it prefers the attribute with a large number of distinct values. For instance, consider an attribute with a unique identifier such as customer\_ID has zero info(D) because of pure partition. This maximizes the information gain and creates useless partitioning.

C4.5, an improvement of ID3, uses an extension to information gain known as the gain ratio. Gain ratio handles the issue of bias by normalizing the information gain using Split Info.

**Gini index**

Another decision tree algorithm CART (Classification and Regression Tree) uses the Gini method to create split points.

The Gini Index considers a binary split for each attribute. You can compute a weighted sum of the impurity of each partition. If a binary split on attribute A partitions data D into D1 and D2, the Gini index of D is:

In case of a discrete-valued attribute, the subset that gives the minimum gini index for that chosen is selected as a splitting attribute. In the case of continuous-valued attributes, the strategy is to select each pair of adjacent values as a possible split-point and point with smaller gini index chosen as the splitting point.The attribute with minimum Gini index is chosen as the splitting attribute.

## Optimizing Decision Tree Performance

* **criterion : optional (default=”gini”) or Choose attribute selection measure**: This parameter allows us to use the different-different attribute selection measure. Supported criteria are “gini” for the Gini index and “entropy” for the information gain.
* **splitter : string, optional (default=”best”) or Split Strategy**: This parameter allows us to choose the split strategy. Supported strategies are “best” to choose the best split and “random” to choose the best random split.
* **max\_depth : int or None, optional (default=None) or Maximum Depth of a Tree**: The maximum depth of the tree. If None, then nodes are expanded until all the leaves contain less than min\_samples\_split samples. The higher value of maximum depth causes overfitting, and a lower value causes underfitting.

**In Scikit-learn, optimization of decision tree classifier performed by only pre-pruning. Maximum depth of the tree can be used as a control variable for pre-pruning. In the following the example, you can plot a decision tree on the same data with max\_depth=3. Other than pre-pruning parameters, You can also try other attribute selection measure such as entropy.**

1. **Decision Tree Classifier Building in Scikit-learn**

### Importing Required Libraries

### # Load libraries

### import pandas as pd

### from sklearn.tree import DecisionTreeClassifier # Import Decision Tree Classifier

### from sklearn.model\_selection import train\_test\_split # Import train\_test\_split function

### from sklearn import metrics #Import scikit-learn metrics module for accuracy calculation

### Loading Data

### #import pandas

### import pandas as pd

### # load dataset

### pima = pd.read\_csv('/Users/user/Desktop/7BUIS008W/diabetes.csv')

### pima.head()

### Feature Selection

#split dataset in features and target variable

feature\_cols = ['Pregnancies', 'Insulin', 'BMI', 'Age','Glucose','BMI','DiabetesPedigreeFunction']

X = pima[feature\_cols] # Features

y = pima.Outcome # Target variable

### Splitting Data

# Split dataset into training set and test set

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.3, random\_state=1) # 70% training and 30% test

### Building Decision Tree Model

### # Create Decision Tree classifer object

### clf = DecisionTreeClassifier()

### # Train Decision Tree Classifer

### clf = clf.fit(X\_train,y\_train)

### #Predict the response for test dataset

### y\_pred = clf.predict(X\_test)

### Evaluating Model

## # Model Accuracy, how often is the classifier correct?

## print("Accuracy:",metrics.accuracy\_score(y\_test, y\_pred))

### Well, you got a classification rate of 67.53%, considered as good accuracy. You can improve this accuracy by tuning the parameters in the Decision Tree Algorithm.

### Optimising Decision Tree Performance

### # Create Decision Tree classifer object

### clf = DecisionTreeClassifier(criterion="entropy", max\_depth=3)

### # Train Decision Tree Classifer

### clf = clf.fit(X\_train,y\_train)

### #Predict the response for test dataset

### y\_pred = clf.predict(X\_test)

### # Model Accuracy, how often is the classifier correct?

### print("Accuracy:",metrics.accuracy\_score(y\_test, y\_pred))

Well, the classification rate increased to 77.05%, which is better accuracy than the previous model.

### Visualizing Decision Trees

### from sklearn import tree

### from matplotlib import pyplot as plt

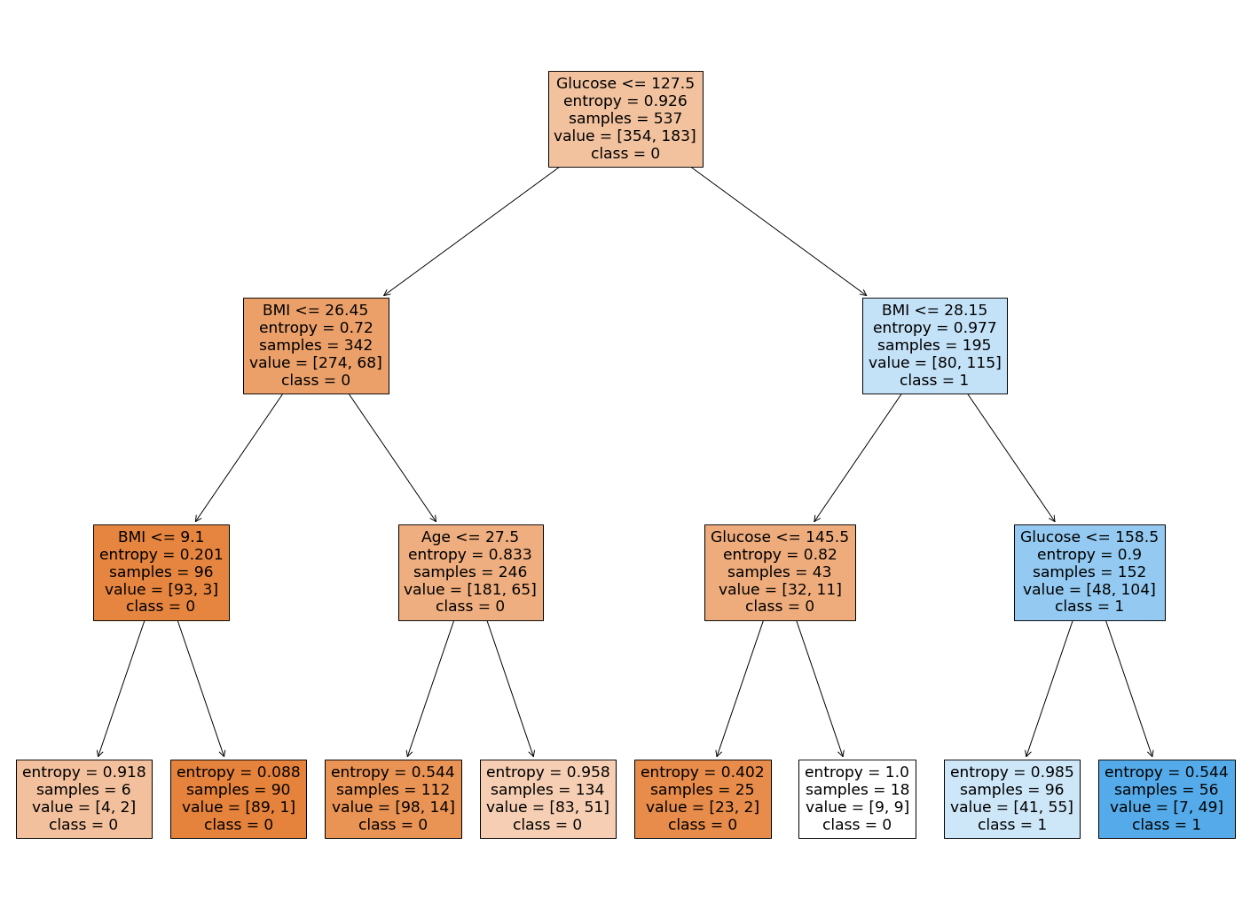
### fig = plt.figure(figsize=(25,20))

### \_ = tree.plot\_tree(clf,

### feature\_names=feature\_cols,

### class\_names=['0','1'],

### filled=True)

****

**Pros**

* Decision trees are easy to interpret and visualize.
* It can easily capture Non-linear patterns.
* It requires fewer data preprocessing from the user, for example, there is no need to normalize columns.
* It can be used for feature engineering such as predicting missing values, suitable for variable selection.
* The decision tree has no assumptions about distribution because of the non-parametric nature of the algorithm.

**Cons**

* Sensitive to noisy data. It can overfit noisy data.
* The small variation(or variance) in data can result in the different decision tree. This can be reduced by bagging and boosting algorithms.
* Decision trees are biased with imbalance dataset, so it is recommended that balance out the dataset before creating the decision tree.

### 2. Decision Tree for Regression

### The process of solving regression problem with decision tree using Scikit Learn is very similar to that of classification. However for regression we use DecisionTreeRegressor class of the tree library. Also the evaluation metrics for regression differ from those of classification. The rest of the process is almost same.

### The dataset we will use for this section can be used with the Linear Regression algorithm. We will use this dataset to try and predict gas consumptions (in millions of gallons) in 48 US states based upon gas tax (in cents), per capita income (dollars), paved highways (in miles) and the proportion of population with a drivers license. The dataset is available on BB:

### Importing Libraries

### import pandas as pd

### import numpy as np

### import matplotlib.pyplot as plt

### #Importing the Dataset

### dataset = pd.read\_csv('D:\Datasets\petrol\_consumption.csv')

### Data Analysis

### We will again use the head function of the dataframe to see what our data actually looks like:

### dataset.head()

### The output looks like this:

|  | Petrol\_tax |  |  | Average\_income | Paved\_Highways | Population\_Driver\_license(%) | Petrol\_Consumption |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 9.0 |  |  | 3571 | 1976 | 0.525 | 541 |
| 1 | 9.0 |  |  | 4092 | 1250 | 0.572 | 524 |
| 2 | 9.0 |  |  | 3865 | 1586 | 0.580 | 561 |
| 3 | 7.5 |  |  | 4870 | 2351 | 0.529 | 414 |
| 4 | 8.0 |  |  | 4399 | 431 | 0.544 | 410 |

### To see statistical details of the dataset, execute the following command:

### dataset.describe()

|  | Petrol\_tax | Average\_income | Paved\_Highways | Population\_Driver\_license(%) | Petrol\_Consumption |
| --- | --- | --- | --- | --- | --- |
| count | 48.000000 | 48.000000 | 48.000000 | 48.000000 | 48.000000 |
| mean | 7.668333 | 4241.833333 | 5565.416667 | 0.570333 | 576.770833 |
| std | 0.950770 | 573.623768 | 3491.507166 | 0.055470 | 111.885816 |
| min | 5.000000 | 3063.000000 | 431.000000 | 0.451000 | 344.000000 |
| 25% | 7.000000 | 3739.000000 | 3110.250000 | 0.529750 | 509.500000 |
| 50% | 7.500000 | 4298.000000 | 4735.500000 | 0.564500 | 568.500000 |
| 75% | 8.125000 | 4578.750000 | 7156.000000 | 0.595250 | 632.750000 |
| max | 10.00000 | 5342.000000 | 17782.000000 | 0.724000 | 986.000000 |

### Preparing the Data

### As with the classification task, in this section we will divide our data into attributes and labels and consequently into training and test sets.Execute the following commands to divide data into labels and attributes:

### X = dataset.drop('Petrol\_Consumption', axis=1)

### y = dataset['Petrol\_Consumption']

### Here the X variable contains all the columns from the dataset, except 'Petrol\_Consumption' column, which is the label.

### The y variable contains values from the 'Petrol\_Consumption' column, which means that the X variable contains the attribute set and y variable contains the corresponding labels.

### Execute the following code to divide our data into training and test sets:

### from sklearn.model\_selection import train\_test\_split

### X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=0)

### Training and Making Predictions

### As mentioned earlier, for a regression task we'll use a different sklearn class than we did for the classification task. The class we'll be using here is the DecisionTreeRegressor class, as opposed to the DecisionTreeClassifier from before.

### To train the tree, we'll instantiate the DecisionTreeRegressor class and call the fit method:

### from sklearn.tree import DecisionTreeRegressor

### regressor = DecisionTreeRegressor()

### regressor.fit(X\_train, y\_train)

### #To make predictions on the test set, ues the predict method:

### y\_pred = regressor.predict(X\_test)

### Now let's compare some of our predicted values with the actual values and see how accurate we were:

### df=pd.DataFrame({'Actual':y\_test, 'Predicted':y\_pred})

### df

### The output looks like this:

|  | Actual | Predicted |
| --- | --- | --- |
| 41 | 699 | 631.0 |
| 2 | 561 | 524.0 |
| 12 | 525 | 510.0 |
| 36 | 640 | 704.0 |
| 38 | 648 | 524.0 |
| 9 | 498 | 510.0 |
| 24 | 460 | 510.0 |
| 13 | 508 | 603.0 |
| 35 | 644 | 631.0 |

### Evaluating the Algorithm

### To evaluate performance of the regression algorithm, the commonly used metrics are mean absolute error, mean squared error, and root mean squared error. The Scikit-Learn library contains functions that can help calculate these values for us. To do so, use this code from the metrics package:

### from sklearn import metrics

### print('Mean Absolute Error:', metrics.mean\_absolute\_error(y\_test, y\_pred))

### print('Mean Squared Error:', metrics.mean\_squared\_error(y\_test, y\_pred))

### print('Root Mean Squared Error:', np.sqrt(metrics.mean\_squared\_error(y\_test, y\_pred)))

### The output should look something like this:

### Mean Absolute Error: 46.8

**Mean Squared Error:** 3850.2

**Root Mean Squared Error:** 62.04997985495241

### The mean absolute error for our algorithm is 46.8, which is less than 10 percent of the mean of all the values in the 'Petrol\_Consumption' column. This means that our algorithm did a fine prediction job.