Decision Trees and Random Forest

AcadView

June 6, 2018

1 Decision tree

1.1 Overview

A decision tree is one of most frequently and widely used supervised machine learning algorithms that can perform both regression and classification tasks. The intuition behind the decision tree algorithm is simple, yet also very powerful.

For each attribute in the dataset, the decision tree algorithm forms a node, where the most important attribute is placed at the root node. For evaluation we start at the root node and work our way down the tree by following the corresponding node that meets our condition or ”decision”. This process continues until a leaf node is reached, which contains the prediction or the outcome of the decision tree.

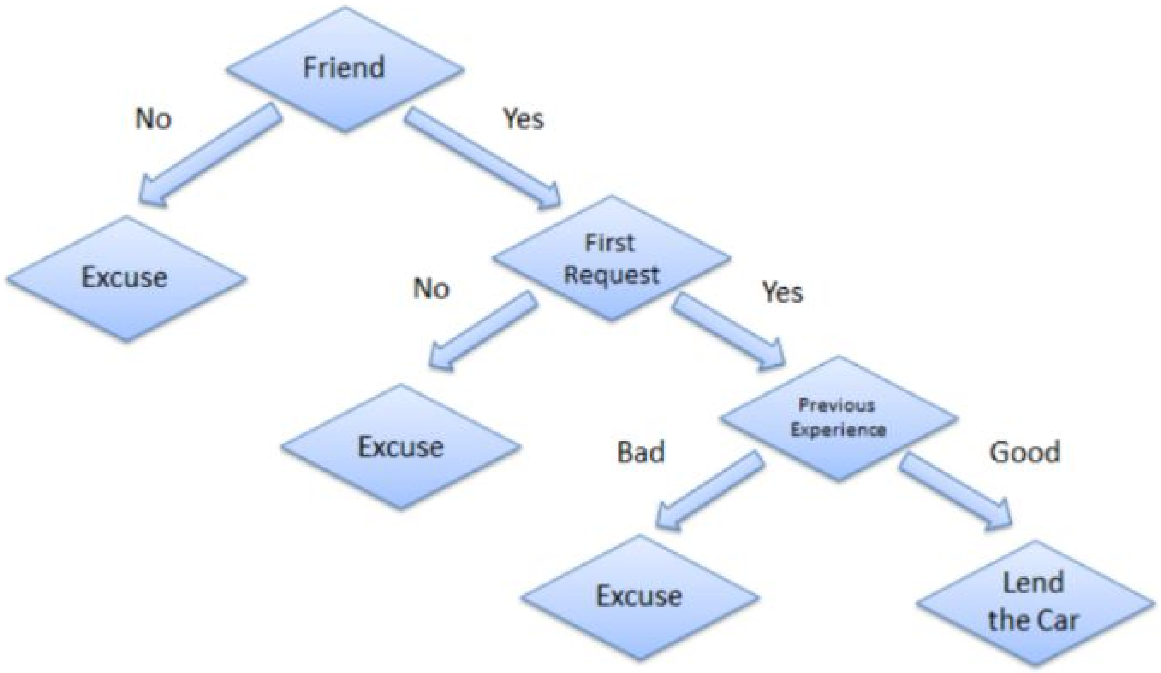
This may sound a bit complicated at first, but what you probably don’t realize is that you have been using decision trees to make decisions your entire life without even knowing it. Consider a scenario where a person asks you to lend them your car for a day, and you have to make a decision whether or not to lend them the car. There are several factors that help determine your decision, some of which have been listed below:

• Is this person a close friend or just an acquaintance? If the person is just an acquaintance, then decline the request; if the person is friend, then move to next step.

• Is the person asking for the car for the first time? If so, lend them the car, otherwise move to next step.

• Was the car damaged last time they returned the car? If yes, decline the request; if no, lend them the car.

The decision tree for the aforementioned scenario looks like this:



The general motive of using Decision Tree is to create a training model which can use to predict class or value of target variables by learning decision rules inferred from prior data(training data).

The understanding level of Decision Trees algorithm is so easy compared with other classification algorithms. The decision tree algorithm tries to solve the problem, by using tree representation. Each internal node of the tree corresponds to an attribute, and each leaf node corresponds to a class label.

1.2 Decision Tree Algorithm Pseudocode

• Place the best attribute of the dataset at the root of the tree.

• Split the training set into subsets. Subsets should be made in such a way that each subset contains data with the same value for an attribute.

• Repeat step 1 and step 2 on each subset until you find leaf nodes in all the branches of the tree.

In decision trees, for predicting a class label for a record we start from the root of the tree. We compare the values of the root attribute with record's attribute. On the basis of comparison, we follow the branch corresponding to that value and jump to the next node.

We continue comparing our record’s attribute values with other internal nodes of the tree until we reach a leaf node with predicted class value. As we know how the modeled decision tree can be used to predict the target class or the value.

1.3 Advantages of Decision Trees

There are several advantages of using decision trees for predictive analysis:

• Decision trees can be used to predict both continuous and discrete values i.e. they work well for both regression and classification tasks.

• They require relatively less effort for training the algorithm.

• They can be used to classify non-linearly separable data.

• They’re very fast and efficient compared to KNN and other classification algorithms.

1.4 Implementing Decision Trees with Python Scikit Learn

In this section we will predict whether a bank note is authentic or fake depending upon the four different attributes of the image of the note. The attributes are Variance of wavelet transformed image, curtosis of the image, entropy, and skewness of the image.

Dataset The dataset for this task can be downloaded from this link:

https://drive.google.com/open?id=13nw-uRXPY8XIZQxKRNZ3yYlho-CYm_Qt

For more detailed information about this dataset, check out the UCI ML repo for this dataset. The rest of the steps to implement this algorithm in Scikit-Learn are identical to any typical machine learning problem, we will import libraries and datasets, perform some data analysis, divide the data into training and testing sets, train the algorithm, make predictions, and finally we will evaluate the algorithm’s performance on our dataset.

Importing Libraries

The following script imports required libraries:

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
```

Importing the Dataset

Since our file is in CSV format, we will use panda's read_csv method to read our CSV data file. Execute the following script to do so:

```
dataset = pd.read_csv("D:/Datasets/bill_authentication.csv")
```

Data Analysis

Execute the following command to inspect the first five records of the dataset:

```
dataset.head()
```

The output will look like this:

	Variance	Skewness	Curtosis	Entropy	Class
0	3.62160	8.6661	-2.8073	-0.44699	0
1	4.54590	8.1674	-2.4586	-1.46210	0
2	3.86600	-2.6383	1.9242	0.10645	0
3	3.45660	9.5228	-4.0112	-3.59440	0
4	0.32924	-4.4552	4.5718	-0.98880	0

Preparing the Data

In this section we will divide our data into attributes and labels and will then divide the resultant data into both training and test sets. By doing this we can train our algorithm on one set of data and then test it out on a completely different set of data that the algorithm hasn't seen yet. This provides you with a more accurate view of how your trained algorithm will actually perform.

To divide data into attributes and labels, execute the following code:

```
X = dataset.drop('Class', axis=1)
y = dataset['Class']
```

Here the X variable contains all the columns from the dataset, except the "Class" column, which is the label. The y variable contains the values from the "Class" column. The X variable is our attribute set and y variable contains corresponding labels.

The final preprocessing step is to divide our data into training and test sets. The model selection library of Scikit-Learn contains train test split method, which we'll use to randomly split the data into training and testing sets. Execute the following code to do so:

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.20)
```

In the code above, the test size parameter specifies the ratio of the test set, which we use to split up 20% of the data in to the test set and 80% for training.

Training and Making Predictions

Once the data has been divided into the training and testing sets, the final step is to train the decision tree algorithm on this data and make predictions. Scikit-Learn contains the tree library, which contains built-in classes/methods for various decision tree algorithms. Since we are going to perform a classification task here, we will use the DecisionTreeClassifier class for this example. The fit method of this class is called to train the algorithm on the training data, which is passed as parameter to the fit method. Execute the following script to train the algorithm:

```
from sklearn.tree import DecisionTreeClassifier
classifier = DecisionTreeClassifier()
classifier.fit(X_train, y_train)
```

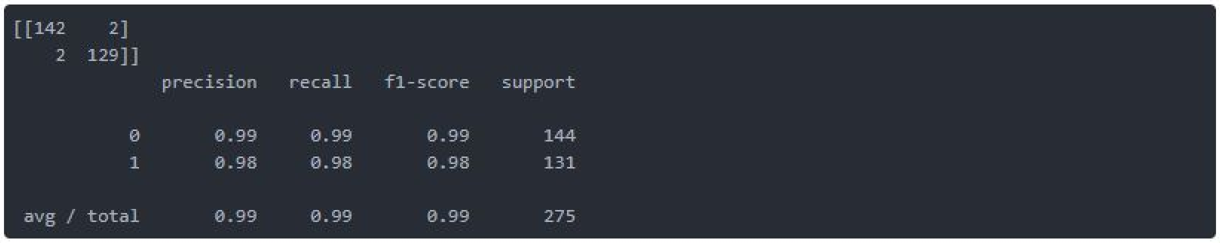
Now that our classifier has been trained, let's make predictions on the test data. To make predictions, the predict method of the DecisionTreeClassifier class is used. Take a look at the following code for usage:

```
y_pred = classifier.predict(X_test)
```

Evaluating the Algorithm At this point we have trained our algorithm and made some predictions. Now we'll see how accurate our algorithm is. For classification tasks some commonly used metrics are confusion matrix, precision, recall, and F1 score. Lucky for us Scikit-Learn's metrics library contains the classification report and confusion matrix methods that can be used to calculate these metrics for us:

```
from sklearn.metrics import classification_report, confusion_matrix
print(confusion_matrix(y_test, y_pred))
print(classification_report(y_test, y_pred))
```

This will produce the following evaluation:



```
[[142    2]
   2  129]]
             precision    recall  f1-score   support

          0       0.99      0.99      0.99       144
          1       0.98      0.98      0.98       131

avg / total       0.99      0.99      0.99       275
```

From the confusion matrix, you can see that out of 275 test instances, our algorithm misclassified only 4. This is 98.5 % accuracy.

2 Random Forests

2.1 Overview

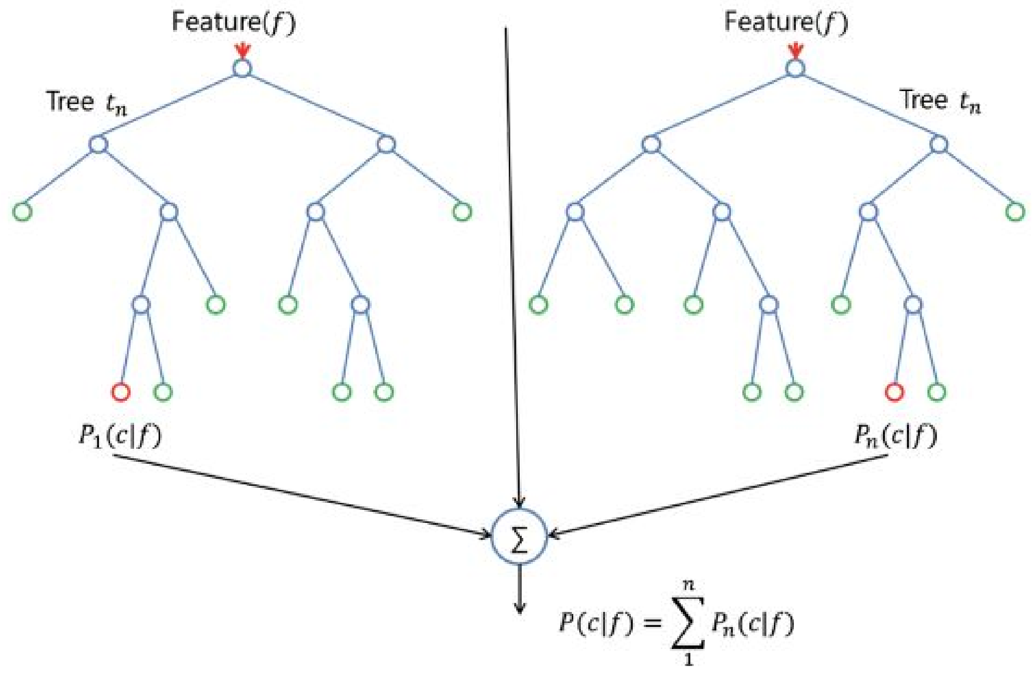
Random Forest is a flexible, easy to use machine learning algorithm that produces, even without hyper-parameter tuning, a great result most of the time. It is also one of the most used algorithms, because it's simplicity and the fact that it can be used for both classification and regression tasks. In this post, you are going to learn, how the random forest algorithm works and several other important things about it.

2.2 How it works

Random Forest is a supervised learning algorithm. Like you can already see from it's name, it creates a forest and makes it somehow random. The forest it builds, is an ensemble of Decision Trees, most of the time trained with the bagging method. The general idea of the bagging method is that a combination of learning models increases the overall result.

To say it in simple words: Random forest builds multiple decision trees and merges them together to get a more accurate and stable prediction. One big advantage of random forest is, that it can be used for both classification and regression problems, which form the majority of current machine learning systems. I will talk about random forest in classification, since classification is sometimes considered the building block of machine learning.

Below you can see how a random forest would look like with two trees:



With a few exceptions a random-forest classifier has all the hyperparameters of a decision-tree classifier and also all the hyperparameters of a bagging classifier, to control the ensemble itself. Instead of building a bagging-classifier and passing it into a decision-tree-classifier, you can just use the random-forest classifier class, which is more convenient and optimized for decision trees. Note that there is also a random-forest regressor for regression tasks.

The random-forest algorithm brings extra randomness into the model, when it is growing the trees. Instead of searching for the best feature while splitting a node, it searches for the best feature among a random subset of features. This process creates a wide diversity, which generally results in a better model.

Therefore when you are growing a tree in random forest, only a random subset of the features is considered for splitting a node. You can even make trees more random, by using random thresholds on top of it, for each feature rather than searching for the best possible thresholds (like a normal decision tree does).

2.3 Random forest classifier pseudocode

To perform prediction using the trained random forest algorithm uses the below pseudocode.

• Takes the test features and use the rules of each randomly created decision tree to predict the outcome and stores the predicted outcome (target)

• Calculate the votes for each predicted target.

• Consider the high voted predicted target as the final prediction from the random forest algorithm.

To perform the prediction using the trained random forest algorithm we need to pass the test features through the rules of each randomly created trees. Suppose let's say we formed 100 random decision trees to from the random forest.

Each random forest will predict different target (outcome) for the same test feature. Then by considering each predicted target votes will be calculated. Suppose the 100 random decision trees are prediction some 3 unique targets x, y, z then the votes of x is nothing but out of 100 random decision tree how many trees prediction is x.

Likewise for other 2 targets (y, z). If x is getting high votes. Let's say out of 100 random decision tree 60 trees are predicting the target will be x. Then the final random forest returns the x as the predicted target. This concept of voting is known as **majority voting.**

2.4 Real Life Analogy:

Imagine a guy named Andrew, that wants to decide, to which places he should travel during a one-year vacation trip. He asks people who know him for advice. First, he goes to a friend, that asks Andrew where he traveled to in the past and if he liked it or not. Based on the answers, he will give Andrew some advice.

This is a typical decision tree algorithm approach. Andrews friend created rules to guide his decision about what he should recommend, by using the answers of Andrew. Afterwards, Andrew starts asking more and more of his friends to advise him and they again ask him different questions, where they can derive some recommendations from. Then he chooses the places that where recommend the most to him, which is the typical Random Forest algorithm approach.

2.5 Implementing Random Forests using scikit-learn

The data for this tutorial is famous. Called, the iris dataset, it contains four variables measuring various parts of iris flowers of three related species, and then a fourth variable with the species name. The reason it is so famous in machine learning and statistics communities is because the data requires very little preprocessing (i.e. no missing values, all features are floating numbers, etc.).

Importing modules

```
# Load the library with the iris dataset
from sklearn.datasets import load_iris

# Load scikit's random forest classifier library
from sklearn.ensemble import RandomForestClassifier

# Load pandas
import pandas as pd

# Load numpy
import numpy as np

# Set random seed
np.random.seed(0)
```

Loading Data

```
# Create an object called iris with the iris data
iris = load_iris()

# Create a dataframe with the four feature variables
df = pd.DataFrame(iris.data, columns=iris.feature_names)

# View the top 5 rows
df.head()
```

	sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)
0	5.1	3.5	1.4	0.2
1	4.9	3.0	1.4	0.2
2	4.7	3.2	1.3	0.2
3	4.6	3.1	1.5	0.2
4	5.0	3.6	1.4	0.2

```
# Add a new column with the species names, this is what we are going to try to predict
df['species'] = pd.Categorical.from_codes(iris.target, iris.target_names)

# View the top 5 rows
df.head()
```

	sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)	species
0	5.1	3.5	1.4	0.2	setosa
1	4.9	3.0	1.4	0.2	setosa
2	4.7	3.2	1.3	0.2	setosa
3	4.6	3.1	1.5	0.2	setosa
4	5.0	3.6	1.4	0.2	setosa

Create Training And Test Data

```
# Create a new column that for each row, generates a random number between 0 and 1, and
# if that value is less than or equal to .75, then sets the value of that cell as True
# and false otherwise. This is a quick and dirty way of randomly assigning some rows to
# be used as the training data and some as the test data.
df['is_train'] = np.random.uniform(0, 1, len(df)) <= .75

# View the top 5 rows
df.head()
```

	sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)	species	is_train
0	5.1	3.5	1.4	0.2	setosa	True
1	4.9	3.0	1.4	0.2	setosa	True
2	4.7	3.2	1.3	0.2	setosa	True
3	4.6	3.1	1.5	0.2	setosa	True
4	5.0	3.6	1.4	0.2	setosa	True

```
# Create two new dataframes, one with the training rows, one with the test rows
train, test = df[df['is_train']==True], df[df['is_train']==False]
```

```
# Show the number of observations for the test and training dataframes
print('Number of observations in the training data:', len(train))
print('Number of observations in the test data:',len(test))
```

```
Number of observations in the training data: 118
Number of observations in the test data: 32
```

Preprocess Data

```
# Create a list of the feature column's names
features = df.columns[:4]

# View features
features
```

```
Index(['sepal length (cm)', 'sepal width (cm)', 'petal length (cm)',
       'petal width (cm)'],
      dtype='object')
```

```
# train['species'] contains the actual species names. Before we can use it,
# we need to convert each species name into a digit. So, in this case there
# are three species, which have been coded as 0, 1, or 2.
y = pd.factorize(train['species'])[0]

# View target
y
```

```
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
       2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
       2, 2, 2])
```

Train the Classifier

```
# Create a random forest Classifier. By convention, clf means 'Classifier'
clf = RandomForestClassifier(n_jobs=2, random_state=0)

# Train the Classifier to take the training features and learn how they relate
# to the training y (the species)
clf.fit(train[features], y)
```

```
RandomForestClassifier(bootstrap=True, class_weight=None, criterion='gini',
            max_depth=None, max_features='auto', max_leaf_nodes=None,
            min_impurity_split=1e-07, min_samples_leaf=1,
            min_samples_split=2, min_weight_fraction_leaf=0.0,
            n_estimators=10, n_jobs=2, oob_score=False, random_state=0,
            verbose=0, warm_start=False)
```

Apply Classifier To Test Data

If you have been following along, you will know we only trained our classifier on part of the data, leaving the rest out. This is, in my humble opinion, the most important part of machine learning. Why? Because by leaving out a portion of the data, we have a set of data to test the accuracy of our model!

Let's do that now.

```
# Apply the Classifier we trained to the test data (which, remember, it has never seen before)
clf.predict(test[features])
```

```
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 2, 2, 1, 1, 2, 2, 2,
       2, 2, 2, 2, 2, 2, 2, 2, 2])
```

What are you looking at above? Remember that we coded each of the three species of plant as 0, 1, or 2. What the list of numbers above is showing you is what species our model predicts each plant is based on the the sepal length, sepal width, petal length, and petal width. How confident is the classifier about each plant? We can see that too.

```
# View the predicted probabilities of the first 10 observations
clf.predict_proba(test[features])[0:10]
```

```
array([[ 1. ,  0. ,  0. ],
       [ 1. ,  0. ,  0. ],
       [ 1. ,  0. ,  0. ],
       [ 1. ,  0. ,  0. ],
       [ 1. ,  0. ,  0. ],
       [ 1. ,  0. ,  0. ],
       [ 1. ,  0. ,  0. ],
       [ 0.9,  0.1,  0. ],
       [ 1. ,  0. ,  0. ],
       [ 1. ,  0. ,  0. ]])
```

There are three species of plant, thus [1. , 0. , 0.] tells us that the classifier is certain that the plant is the first class. Taking another example, [0.9, 0.1, 0.] tells us that the classifier gives a 90% probability the plant belongs to the first class and a 10% probability the plant belongs to the second class. Because 90 is greater than 10, the classifier predicts the plant is the first class.

Evaluate Classifier

Now that we have predicted the species of all plants in the test data, we can compare our predicted species with the that plants actual species.

```
# Create actual english names for the plants for each predicted plant class
preds = iris.target_names[clf.predict(test[features])]
```

```
# View the PREDICTED species for the first five observations
preds[0:5]
```

```
array(['setosa', 'setosa', 'setosa', 'setosa', 'setosa'],
      dtype='<U10')
```

```
# View the ACTUAL species for the first five observations
test['species'].head()
```

```
7     setosa
8     setosa
10    setosa
13    setosa
17    setosa
Name: species, dtype: category
Categories (3, object): [setosa, versicolor, virginica]
```

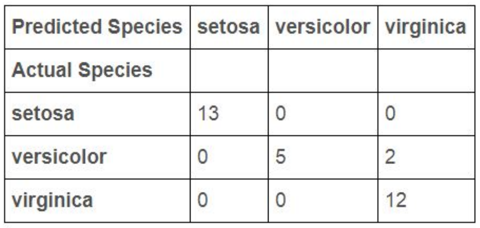
That looks pretty good! At least for the first five observations. Now let's use look at all the data.

Create a confusion matrix

A confusion matrix can be, no pun intended, a little confusing to interpret at first, but it is actually very straightforward. The columns are the species we predicted for the test data and the rows are the actual species for the test data. So, if we take the top row, we can see that we predicted all 13 setosa plants in the test data perfectly. However, in the next row, we predicted 5 of the versicolor plants correctly, but mis-predicted two of the versicolor plants as virginica.

The short explanation of how to interpret a confusion matrix is: anything on the diagonal was classified correctly and anything off the diagonal was classified incorrectly.

```
# Create confusion matrix
pd.crosstab(test['species'], preds, rownames=['Actual Species'], colnames=['Predicted Species'])
```



Predicted Species	setosa	versicolor	virginica
Actual Species			
setosa	13	0	0
versicolor	0	5	2
virginica	0	0	12