
**Spring 2020**

Data Mining taught by Dr.Assem Nasser

CSC 5345- 01

# Report: Decision Tree Implementation

Hamza Touhs

Submission Date: 10/04/2020

**Decision Tree Algorithm:**

A decision tree is a flowchart-like tree structure where an internal node represents feature/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 recursively. This flowchart-like structure helps you in decision making. It's visualization like a flowchart diagram which easily mimics the human thinking process.

**Dataset:**

In this project I will be using the Pima Indians Diabetes Dataset which is available in the National Institute of Diabetes and Digestive and Kidney Diseases. This implementation involves classifying the onset of diabetes (binary) within 5 years in Pinma Indians given medical details. The number of observations for each class is not balanced. There are 768 observations with 8 input variables and 1 output variable. Missing values are encoded with zero values. The attributes are:

1. Number of times pregnant.
2. Plasma glucose concentration 2 hours in an oral glucose tolerance test.
3. Diastolic blood pressure (mm Hg).
4. Triceps skinfold thickness (mm).
5. 2-Hour serum insulin (mu U/ml).
6. Body mass index (weight in kg/(height in m)^2).

7. Diabetes pedigree function.
8. Age (years).
9. Class variable (0 or 1).

In competitive websites the baseline performance of predicting the most prevalent class is a classification accuracy of approximately 65%. Top results achieve a classification accuracy of approximately 77%.

A sample of the first 5 rows is listed below:

```
6,148,72,35,0,33.6,0.627,50,1
1,85,66,29,0,26.6,0.351,31,0
8,183,64,0,0,23.3,0.672,32,1
1,89,66,23,94,28.1,0.167,21,0
0,137,40,35,168,43.1,2.288,33,1
```

**Implementation:**

After our conversation, I made sure to make a program that can easily run on various datasets, handles categorical and continuous data, and most importantly calculates the accuracy.

We start by asking the user for the path of the dataset. I have included the dataset above in order to test the tree.

```
# load the dataset
filename = 'dataset.txt'
dataset = load_csv(filename)

ds = pd.read_csv(filename)
```

You can notice that I opened the dataset twice. The last line will provide us an instantiation of the file with some interesting operations that are going to be used in order to figure out if a feature is categorical or continuous.

To determine the type of features, we generate a parallel array (parallel to the features) which holds whenever a feature is continuous or categorical. This array is generated by traversing the data columns by columns and each time counting the number of unique values and checking if the values are numbers and if the length of unique attribute is greater than the threshold. In those cases we append “categorical” otherwise we append “continuous”. In order to set the threshold, I tried different values for different datasets; numbers between 5-10 seem to give good results. I tried to set the threshold relative to the other attributes but this would be dataset specific and would have a hard time to generalize to other possible datasets. In order to check whenever a value contains numbers or strings, I used the function float() which is supposed to convert a string that contains numbers into a float. What is interesting is that this functions throws an exception if the inputted string does not contain a numerical value; hence, if the exception happens we can safely assume that the string is made of characters.

```
def isNumber(s):
        try:
        float(s)
        return True
        except ValueError:
        return False
```

Next, because python reads data as strings, we need to convert the numerical values into floats, to achieve this, we traverse our dataset to make use of the isNumber function to check if the value is convertible.

```
def floatify(dataset, column):
    for row in dataset:
        if(not isNumber(row[column])):
            continue
        row[column] = float(row[column].strip())
```

To access the accuracy of our algorithm, we are using cross-validation. To do this we basically need to define k, which represents the number of folds. This is the number of splits that we will do our dataset. Note that "split" in this case is different from the one related to decision trees. We start by calling the function getSplits(). In this function we append folds with random rows until we reach the fold size number which is dataset size over the number of folds. After completing a fold, we append it to the final result array which contains k arrays/folds.

```
def getSplits(dataset, k_folds):
    datasetAfterSplit = list()
    datasetCopy = list(dataset)
    foldSize = int(len(dataset) / k_folds)
    for i in range(k_folds):
        fold = list()
        while len(fold) < foldSize:
            index = randrange(len(datasetCopy))
            fold.append(datasetCopy.pop(index))
        datasetAfterSplit.append(fold)
    return datasetAfterSplit
```

Now we need to use each fold as a test set and remaining as training sets and during each iteration we build a tree, use it to generate predictions, and then calculate its accuracy, by comparing it to the actual value; we then append the score of each fold into an array which we return and use in order to show the scores of the different folds as well as the mean accuracy.

Building the tree and making predictions:

In this phase we need both a train set and a test set. The training set is going to be used to generate the tree andthe test set is used to predict labels which are then going to be used to determine the accuracy of the fold. The prediction phase consists of traversing the tree; each time going right or left based on the node value and most important based on whether it is a categorical value or continuous value. The difference resides in the following: if it is a categorical value, we compare to the actual value (using ==), otherwise, we compare if the value is < than the value picked (we will talk more about those picked values below).

```
def buildAndPredict(train, test):
    tree = buildTree(train)
    predictions = list()
    for row in test:
            prediction = predict(tree, row)
            predictions.append(prediction)
    return(predictions)
```

To build the tree we look for the best split using the function (getBestSplit), we store the returned node in a root and call a split() which mainly calls getBestSplit recursively in order to build the rest of the tree. In this function, we decide whether to continue generating for splits or finish into

a leaf node. To generate more decision nodes, we call the getBestSplit() which we will talk about in more details bellow, and to finish into a leaf node we call leafify() which decides on the prediction which uses a simple voting mechanism by getting the highest number of a class in a side after a split.

```
def leafify(distribution):
    outcomes = [row[-1] for row in distribution]
    return max(set(outcomes), key=outcomes.count)
```

We stop splitting if one of the sides is empty i.e: having pure groups or reached the maximum depth:

```
def split(node, depth):
    left, right = node['distribution']
    # if is the sides are empty, we
    if not left or not right:
            node['left'] = node['right'] = leafify(left + right)
            return
    # check if we reach the max depth
    if depth >= max_depth:
            node['left'], node['right'] = leafify(left), leafify(right)
            return
    # left child If still didnt reach the minimum size we find the new best split and append it
using the split()

    node['left'] = getBestSplit(left)
    split(node['left'], depth+1)
    # right child

    node['right'] = getBestSplit(right)
    split(node['right'], depth+1)
```

In the getbestSplit() we select the best split point for a dataset based on the gini index. We first extract the labels and then initialize some variables where we are storing the columns index, the equivalent value (referred to earlier), the gini index of the split (weighted gini index), and the actual distribution of rows (left/right). We traverse the columns and in each column we check the feature type of that column, if it is continuous we split based on continuous attribute and we use the equivalent function for the categorical features. Similarly, both functions append rows to either left or right based on a condition. This condition is slightly based on the feature type.

```
def getBestSplit(dataset):
   labels = list(set(row[-1] for row in dataset)) #get the last item of the list (the label)
   bestIndex, bestValue, bestScore, bestDistribution = 99999, 99999, 99999, None #starting
point. Typically, they will be replace after the very first split because 0<=gini<=0.5<<999
   for columnIndex in range(len(dataset[0])-1): #Here we skipped the -1 because those are the
labels
         for row in dataset:
               if(feautureTypes[columnIndex] == "continous"):
                     distribution = splitBasedOnContinuousAttribute(columnIndex,
row[columnIndex], dataset) #Split on row[columnIndex] [left,right] (of the tree)
               elif(feautureTypes[columnIndex] == "categorical"):
                     distribution = splitBasedOnCategoricalAttribute(columnIndex,
row[columnIndex], dataset)

               gini = computeGiniIndex(distribution, labels)
               if gini < bestScore:
                     bestIndex, bestValue, bestScore, bestDistribution = columnIndex,
row[columnIndex], gini, distribution
   return {'index':bestIndex, 'value':bestValue, 'distribution':bestDistribution} #We dont need
to pass the score because it
```

When getting the left and right side of that particular split, we need to contify how good the split is. We do that by calculating the gini index. The smaller it is, the better the split is. You can

notice in the computeGiniIndex() that there is a safety check that ensures that we are not going to divide by 0. When traversing each side, we contribute into the generation of the weighted gini index which represents how good the actual split is. If the returned fini index is better than the ones before, we replace the variables we referred to earlier with the new values. AFter iterating through the whole columns, we return the best values that are appended to the tree. When traversing the tree to predict, we will be using the best columns and the best value to split our dataset.

Execution:

To run the program we first need to download the pandas library in case it is not already present. We use this library in order to open the csv file and get the unique values which are used to determine the type of a feature.

Note that because we are using the random library to generate the folds, we can make use of seed() to have consistent outputs. To download the libraries I recommend using pip which is the default package manager of python.

```
pip install pandas
```

To run the program use:

```
Python decisionTree.py (on Windows) and python3 decisionTree.py (on Mac and Linux)
```

The program is going to ask you to enter the name of the dataset, let's first try with the Diabete dataset referred to above as well other datasets (notorious titanic dataset, and the tennisData used in my initial submission). I included the dataset in the .rar file. It is named dataset.txt. Here is the output of the program using three different:

```
C:\Users\touhs\OneDrive\Desktop>python decisionTree.py.py
Dataset Path (or name if file is in the same directory):dataset.txt
Scores: [81.69934640522875, 72.54901960784314, 80.3921568627451, 73.20261437908496, 73.8562091503268]
Mean Accuracy: 76.340%

C:\Users\touhs\OneDrive\Desktop>python decisionTree.py.py
Dataset Path (or name if file is in the same directory):titanic.csv
Scores: [79.21348314606742, 82.58426966292134, 80.33707865168539, 79.7752808988764, 81.46067415730337]
Mean Accuracy: 80.674%

C:\Users\touhs\OneDrive\Desktop>python decisionTree.py.py
Dataset Path (or name if file is in the same directory):tennisData.csv
Scores: [100.0, 97.22222222222221, 100.0, 100.0, 100.0]
Mean Accuracy: 99.444%
```

The first output refers to the scores generated by each of the folds and the second output is the mean accuracy of those folds.

The input files should work on all datasets; however, there are some conditions to meet before feeding the dataset to this program. The dataset must have labels in the very last column and the program will generate better results if relevant features are kept; i.e: dropping names, IDs, etc...

References:

When it came to distinguishing between categorical and continuous attributes, my implement was inspired by: https://www.youtube.com/watch?v=5eoFajw8TWk&t=271s

You can notice that he used it differently, this is because our base implementations are different and the libraries used to open the files are also different. I opted for pandas, as I feel more comfortable manipulating files than the native open_csv() he used.