MARMARA UNIVERSITY
FACULTY OF ENGINEERING

COMPUTER ENGINEERING DEPARTMENT

CSE 3000 Summer Internship Report

(20 Work Days)

Student Name-Surname: Mert KELKİT
Student Number: 150115013
Starting Date: 23/07/2018
Ending Date: 17/08/2018

1. Technologies Used in the Training

During my training, I used the technologies:

- RStudio

RStudio is a free and open-source integrated development environment for R, a programming language for statistical computing and graphics. It provides strong visualization tools, a terminal, variable and file explorer, basically whatever a data scientist needs.

I used RStudio for my R implementations, which are some metrics for my training project. RStudio helped me make these implementations efficiently.

- PyCharm

PyCharm is one of the most used and famous integrated development environment for Python scripting language. It is developed by the Czech company JetBrains. It provides code analysis, a graphical debugger, an integrated unit tester, integration with version control systems.

I choose this IDE because I implement my project mainly with Python, this IDE was familiar to me and made me do my job efficiently.

- Anaconda

Anaconda is a free and open source distribution of the Python and R programming languages for data science and machine learning related applications (large-scale data processing, predictive analytics, scientific computing), that aims to simplify package management and deployment. Package versions are managed by the package management system conda.

I used Anaconda's Python and R interpreters in my training project. It includes useful libraries like scipy, pandas, numpy, scikit-learn, matplotlib for Python; dply, tidyr, ggplot2 for R.

- Graphviz

Graphviz (short for Graph Visualization Software) is a package of open-source tools initiated by AT&T Labs Research for drawing graphs specified in DOT Language scripts. It also provides libraries for software applications to use the tools. Graphviz is free software licensed under the Eclipse Public License.

I used Graphviz library for Python in my project. A Python script generates constructed graph's corresponding DOT Language implementation, then Graphviz library interprets and converts the Python-generated DOT file to a specified formatted file e.g. jpg, pdf.

- Orange

Orange is a component-based visual programming software package for data visualization, machine learning, data mining and data analysis.

Orange components are called widgets and they range from simple data visualization, subset selection and preprocessing, to empirical evaluation of learning algorithms and predictive modeling.

I used Orange application to compare my project's performance with an on hand Orange model's performance. I also compared my visualizations with Orange's visualizations.

- GitHub

GitHub Inc. is a web-based hosting service for version control using Git. It is mostly used for computer code. It offers all of the distributed version control and source code management (SCM) functionality of Git as well as adding its own features. It provides access control and several collaboration features such as bug tracking, feature requests, task management, and wikis for every project.

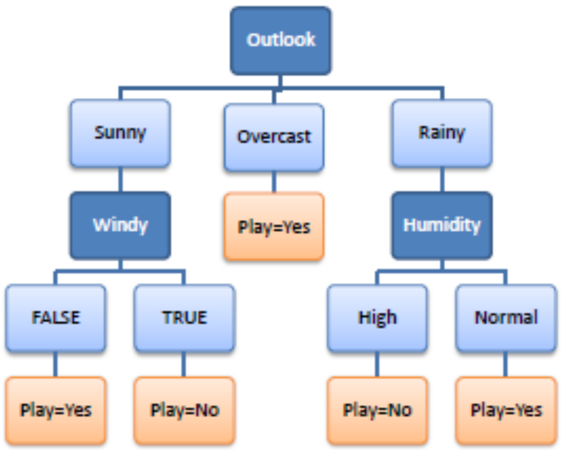
I used GitHub in my training's almost every step. I used it mostly for keeping track of my project's versions with a repository I opened. Also it gave me the opportunity to inspect another project's which are related to mine.

- Jupyter Notebook

The Jupyter Notebook is an open-source web application that allows you to create and share documents that contain live code, equations, visualizations and narrative text. Uses include: data cleaning and transformation, numerical simulation, statistical modeling, data visualization, machine learning, and much more.

I used Jupyter Notebook for visualization, doing small experiments with some metrics and project components because of its ease of use for small code snippets.

2. Work Done



During my training, my project was to implement a fully equipped decision tree for classification problem and visualization of it from scratch. It was expected to give better results than an on hand decision tree classifier e.g. scikit-learn's decision tree. I implemented my data structure mainly with Python, using the PyCharm IDE.

I will describe my weekly progress on decision tree project below:

a. Week 1

At the beginning of my training, I met with the team members then asked them which technologies they are using and what they are doing in a normal working day. After I got my answers, I got username and password for the company's computer and the wifi to begin to work. Then I started learning and practicing SQL - they use Oracle SQL as I mentioned - from interactive web applications like www.w3schools.com or sqlbolt.com.

After a meeting with my supervisor, I have 3 options for my training project which are implementing a linear regression model, implementing a logistic regression model and implementing a decision tree model. I have very little experiences with all of these models and in my opinion, decision tree is the most interesting one. I started researching these 3 models with their logics, usage fields, metrics-calculations behind them and pros-cons. As I wanted at first, I choose implementing a decision tree from scratch. As soon as I choose decision tree classifier for my project, I started researching deeper.

I learned there are some algorithms for decision tree classification. Remarkable ones are "ID3" and "C4.5".

- ID3

The core algorithm for building decision trees called ID3 by J. R. Quinlan which employs a top-down, greedy search through the space of possible branches with no backtracking. ID3 uses Entropy and Information Gain to construct a decision tree. ID3 does not guarantee an optimal solution; it can get stuck in local optima.

It uses a greedy approach by selecting the best attribute to split the dataset on each iteration. One improvement that can be made on the algorithm can be to use backtracking during the search for the optimal decision tree. Also it's hard to handle continuous attributes with this algorithm. If the values of any given attribute is continuous, then there are many more places to split the data on this attribute, and searching for the best value to split by can be time consuming.

- C4.5

C4.5 builds decision trees from a set of training data in the same way as ID3, using the concept of information entropy. The training data is a set $S=s_1, s_2, \ldots$ of already classified samples. Each sample s_i consists of a p-dimensional vector $(x_{1,i}, x_{2,i}, \ldots, x_{p,i})$, where x_j the represent attribute values or features of the sample, as well as the class in which s_i falls.

At each node of the tree, C4.5 chooses the attribute of the data that most effectively splits its set of samples into subsets enriched in one class or the other. The splitting criterion is the normalized information gain (difference in entropy). The attribute with the highest normalized information gain is chosen to make the decision. The C4.5 algorithm then recurs on the smaller sublists.

C4.5 has a number of improvements on ID3 algorithm. There are:

- Handling both continuous and discrete attributes - In order to handle continuous attributes, C4.5 creates a threshold and then splits the list into those whose attribute value is above the threshold and those that are less than or equal to it.
- Handling training data with missing attribute values - C4.5 allows attribute values to be marked as ? for missing. Missing attribute values are simply not used in gain and entropy calculations.
- Pruning trees after creation - C4.5 goes back through the tree once it's been created and attempts to remove branches that do not help by replacing them with leaf nodes.

My algorithm was a C4.5-like algorithm since it can handle both discrete and continuous data - but from a different way. My algorithm also does pre and post pruning both.

Then I found out the most used metrics. They are entropy - information gain (they can be used together) and Gini index.

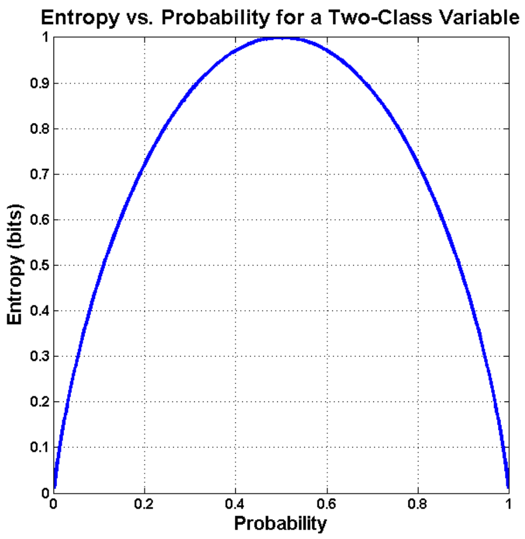
- Entropy and Information Gain

Entropy is used to measure the amount of impurity of a dataset. If the dataset is pure (all of the observations belong to one class), entropy will be 0. If a dataset is not pure - e.g. for a binary classification problem, 50% of observations belong to class 1, 50% of observations belong to class 2 entropy will be 1 in that case -, entropy will be 1 or close to 1. As we can see, lower the entropy, lower the impurity which is a case that we want. Entropy has the formula:

$$H = -\sum_{i=1}^{n} p_i \cdot \log_2(p_i)$$

There are n classes, p_i is the relative frequency of the observations which are belong to class i (i=1..n).

Graphic of entropy function for binary classification problem:



As we can see from the graphic, if we cannot get any knowledge from the split, e.g. after splitting, relative frequencies of observations' classes are equal, entropy will be 1. If after split data is pure, entropy will be 0.

Information gain is used to measure how much information we get from a split. It’s basically [Base Entropy - After Split Entropy]. Unlike entropy, we want higher information gain in our calculations.

Consider an example dataset without any split:

Predictors **Target**

Outlook	Temp.	Humidity	Windy	Play Golf
Rainy	Hot	High	False	No
Rainy	Hot	High	True	No
Overcast	Hot	High	False	Yes
Sunny	Mild	High	False	Yes
Sunny	Cool	Normal	False	Yes
Sunny	Cool	Normal	True	No
Overcast	Cool	Normal	True	Yes
Rainy	Mild	High	False	No
Rainy	Cool	Normal	False	Yes
Sunny	Mild	Normal	False	Yes
Rainy	Mild	Normal	True	Yes
Overcast	Mild	High	True	Yes
Overcast	Hot	Normal	False	Yes
Sunny	Mild	High	True	No

Base entropy calculated as shown below:

Play Golf	
Yes	No
9	5

Entropy(PlayGolf) = Entropy (5,9)

= Entropy (0.36, 0.64)

$= -(0.36 \log_2 0.36) - (0.64 \log_2 0.64)$

= 0.94

Suppose we split our dataset from the column “Outlook”. Then we get a result like this:

		Play Golf		
		Yes	No	
Outlook	Sunny	3	2	5
	Overcast	4	0	4
	Rainy	2	3	5
				14

E(PlayGolf, Outlook) = **P**(Sunny)***E**(3,2) + **P**(Overcast)***E**(4,0) + **P**(Rainy)***E**(2,3)

= (5/14)*0.971 + (4/14)*0.0 + (5/14)*0.971

= 0.693

There are 3 possible splits with the column “Outlook”.

We calculate entropy of each possible Outlook value’s, multiply entropies by their proportion in total, then sum all of the results. That’s the entropy value for splitting dataset from “Outlook” column.

To get information gain from that split, we subtract after split entropy from the base entropy. In that case, splitting from “Outlook” column gives us 0.94 - 0.693 = 0.247 information gain. We can apply this process to all possible splits, then append as a split node to our decision tree with the split that has the maximum information gain - or larger than a threshold value -.

All possible splits’ information gains:

		Play Golf	
		Yes	No
Outlook	Sunny	3	2
	Overcast	4	0
	Rainy	2	3
Gain = 0.247			

		Play Golf	
		Yes	No
Temp.	Hot	2	2
	Mild	4	2
	Cool	3	1
Gain = 0.029			

		Play Golf	
		Yes	No
Humidity	High	3	4
	Normal	6	1
Gain = 0.152			

		Play Golf	
		Yes	No
Windy	False	6	2
	True	3	3
Gain = 0.048			

- Gini Index

Gini Index is a metric to measure how often a randomly chosen element would be incorrectly identified. It means an attribute with lower Gini Index should be preferred. Gini Index has the formula:

$$Gini = 1 - \sum_{i=1}^{C} (p_i)^2$$

Where p is the relative frequency of observations’ classes after split.

Consider these columns. First column is a variable, last column is the target column.

0.2	positive
0.2	positive
0.4	positive
0.2	positive
0.2	positive
0.2	positive
0.2	positive
0.4	positive
1.4	negative
1.5	negative
1.5	negative
1.3	negative
1.5	negative
1.3	negative
1.6	negative
1	negative

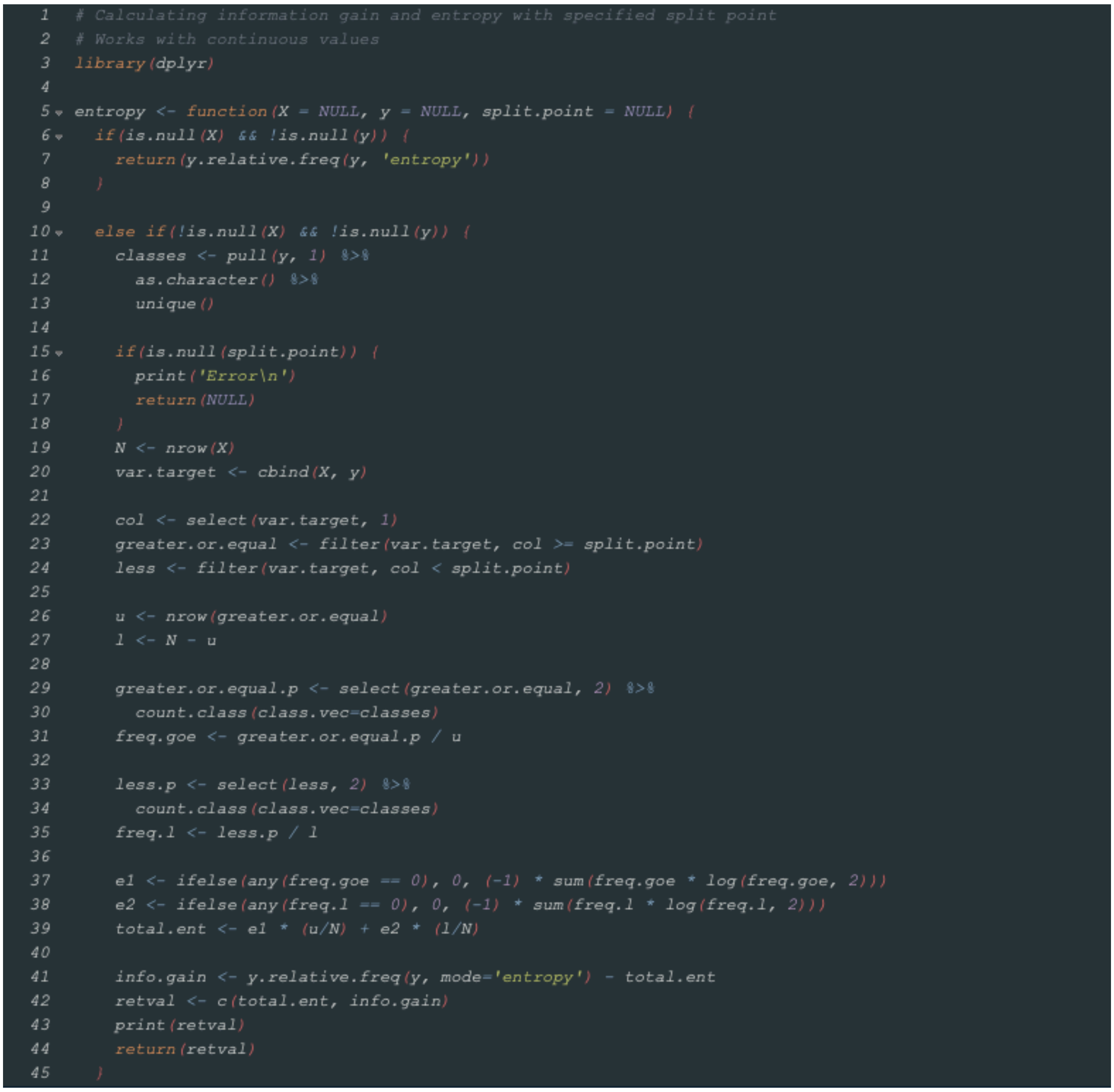
If we split the variable from split point 1.4, target results and Gini Index value will be like this.

		Target	
		Positive	Negative
D	>= 1.4	0	5
	< 1.4	8	3
Gini Index of D= 0.273			

First, we find Gini(0, 5) = 0, then we find Gini(8, 3) = 0.397. Then we multiply these values with their proportions. Total Gini Index is (5/16) * 0 + (11/16) * 0.397 = 0.273.

As soon as I've learned these metrics, I implemented them with R in order to test them, then choose the best metric to implement my first decision tree.

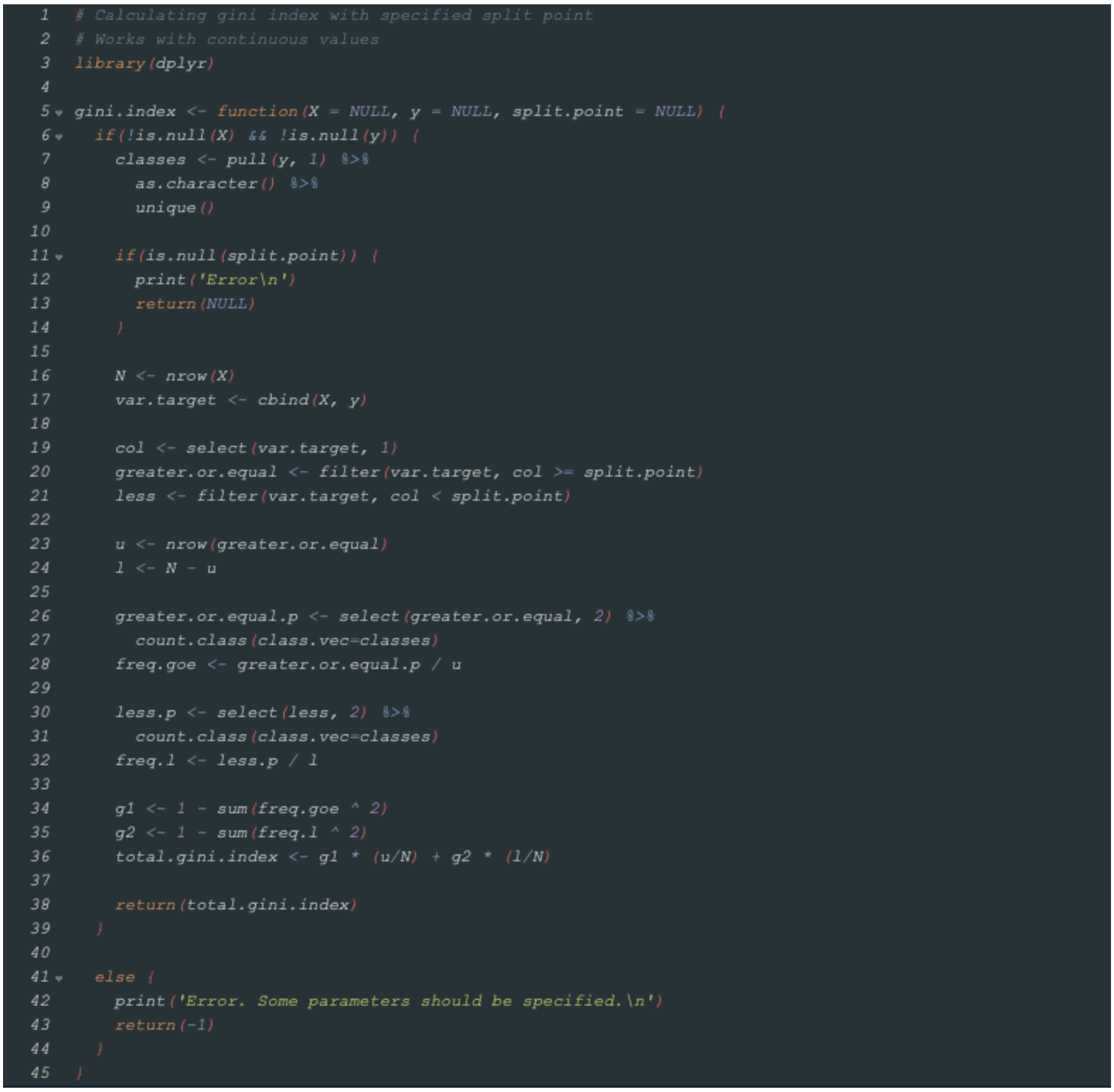
My entropy and information gain implementation:



```r
# Calculating information gain and entropy with specified split point
# Works with continuous values
library(dplyr)

entropy <- function(X = NULL, y = NULL, split.point = NULL) {
  if(is.null(X) && !is.null(y)) {
    return(y.relative.freq(y, 'entropy'))
  }

  else if(!is.null(X) && !is.null(y)) {
    classes <- pull(y, 1) %>%
      as.character() %>%
      unique()

    if(is.null(split.point)) {
      print('Error\n')
      return(NULL)
    }
    N <- nrow(X)
    var.target <- cbind(X, y)

    col <- select(var.target, 1)
    greater.or.equal <- filter(var.target, col >= split.point)
    less <- filter(var.target, col < split.point)

    u <- nrow(greater.or.equal)
    l <- N - u

    greater.or.equal.p <- select(greater.or.equal, 2) %>%
      count.class(class.vec=classes)
    freq.goe <- greater.or.equal.p / u

    less.p <- select(less, 2) %>%
      count.class(class.vec=classes)
    freq.l <- less.p / l

    e1 <- ifelse(any(freq.goe == 0), 0, (-1) * sum(freq.goe * log(freq.goe, 2)))
    e2 <- ifelse(any(freq.l == 0), 0, (-1) * sum(freq.l * log(freq.l, 2)))
    total.ent <- e1 * (u/N) + e2 * (l/N)

    info.gain <- y.relative.freq(y, mode='entropy') - total.ent
    retval <- c(total.ent, info.gain)
    print(retval)
    return(retval)
  }
```

This function expects three inputs, X(independent variable), y(target variable) and split point if X is continuous, to discretize the data. If X is null and y is not null, it means user wants to get base entropy. If both not null, it means user wants to get information gain and entropy after a split operation. I applied the entropy and information gain formulas discussed above in the rest of the code.

My Gini Index implementation:

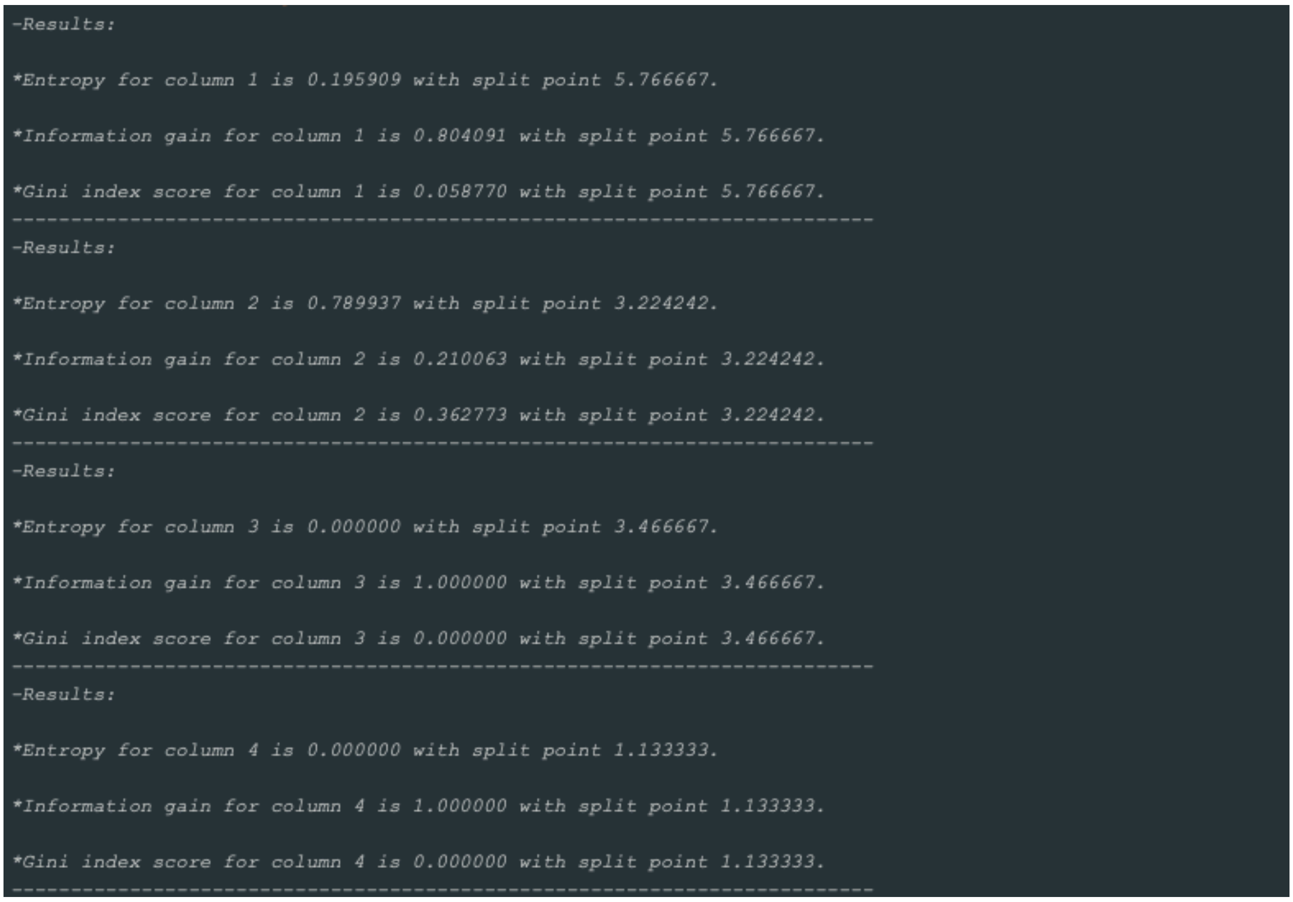


```r
# Calculating gini index with specified split point
# Works with continuous values
library(dplyr)

gini.index <- function(X = NULL, y = NULL, split.point = NULL) {
  if(!is.null(X) && !is.null(y)) {
    classes <- pull(y, 1) %>%
      as.character() %>%
      unique()

    if(is.null(split.point)) {
      print('Error\n')
      return(NULL)
    }

    N <- nrow(X)
    var.target <- cbind(X, y)

    col <- select(var.target, 1)
    greater.or.equal <- filter(var.target, col >= split.point)
    less <- filter(var.target, col < split.point)

    u <- nrow(greater.or.equal)
    l <- N - u

    greater.or.equal.p <- select(greater.or.equal, 2) %>%
      count.class(class.vec=classes)
    freq.goe <- greater.or.equal.p / u

    less.p <- select(less, 2) %>%
      count.class(class.vec=classes)
    freq.l <- less.p / l

    g1 <- 1 - sum(freq.goe ^ 2)
    g2 <- 1 - sum(freq.l ^ 2)
    total.gini.index <- g1 * (u/N) + g2 * (l/N)

    return(total.gini.index)
  }

  else {
    print('Error. Some parameters should be specified.\n')
    return(-1)
  }
}
```

This function contains split operations and applied formula of Gini Index.

I compare results of these two metrics in order to choose one between them, I realized that almost every case, they all came to the same result . After asking my supervisor which metric is more reliable, he said it doesn’t matter, I conclude that it doesn’t matter which metric I use. They are all builded for same purpose. Measure impurity in the data.

These are the results of these two metrics with iris dataset:



```
-Results:

*Entropy for column 1 is 0.195909 with split point 5.766667.

*Information gain for column 1 is 0.804091 with split point 5.766667.

*Gini index score for column 1 is 0.058770 with split point 5.766667.
------------------------------------------------------------------------
-Results:

*Entropy for column 2 is 0.789937 with split point 3.224242.

*Information gain for column 2 is 0.210063 with split point 3.224242.

*Gini index score for column 2 is 0.362773 with split point 3.224242.
------------------------------------------------------------------------
-Results:

*Entropy for column 3 is 0.000000 with split point 3.466667.

*Information gain for column 3 is 1.000000 with split point 3.466667.

*Gini index score for column 3 is 0.000000 with split point 3.466667.
------------------------------------------------------------------------
-Results:

*Entropy for column 4 is 0.000000 with split point 1.133333.

*Information gain for column 4 is 1.000000 with split point 1.133333.

*Gini index score for column 4 is 0.000000 with split point 1.133333.
------------------------------------------------------------------------
```

As we can see, split points were chosen same with these metrics. If we sort their goodness of split, we would get the same order for these metrics.

Split point for a column were chosen by splitting a column from every possible value (every distinct value in the column), then test them with specified metric. But this split point selection method would be too slow for a big dataset, there may be much more distinct values in a column. I used this selection method just for testing metrics, not for whole decision tree.

My first week on training was spent with meeting, lots of researching and testing my implementations with different datasets.

b. Week 2

In week 2, I started implementing my decision tree classifier data structure with Python. First version of my project can handle both nominal and numerical(continuous) data. I chose entropy and information gain as my main metrics for construction of the tree.

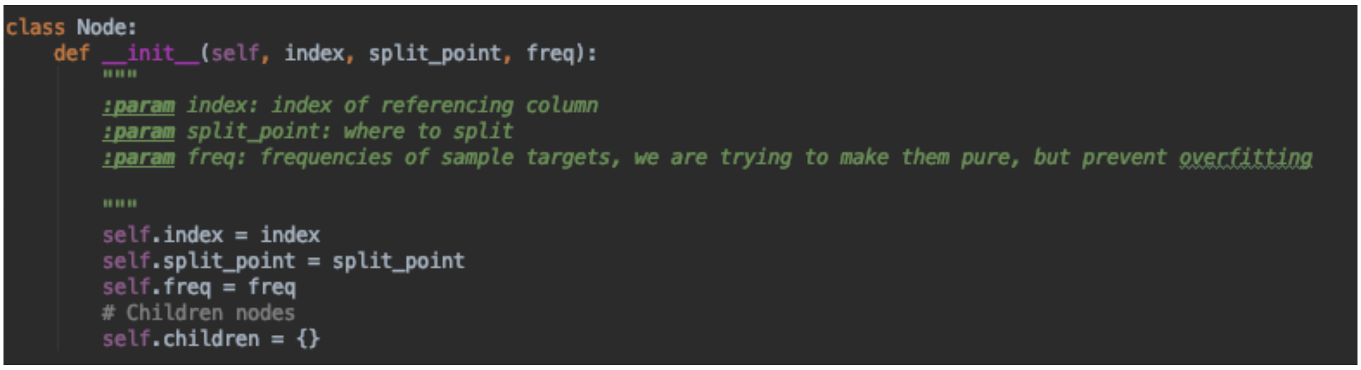
If a variable is nominal(categorical), my algorithm splits the data from all possible categories. For example if a categorical column has 3 different values, there will be 3 branches on the decision tree.

If a variable is numerical, algorithm finds the best split point for this variable in order to discretize the continuous variable. In week 2, I could only implement binary split for continuous variables, e.g. 2 branches are >= [split_point] and <[split_point].

For my data structure, I created 3 classes:

- Node

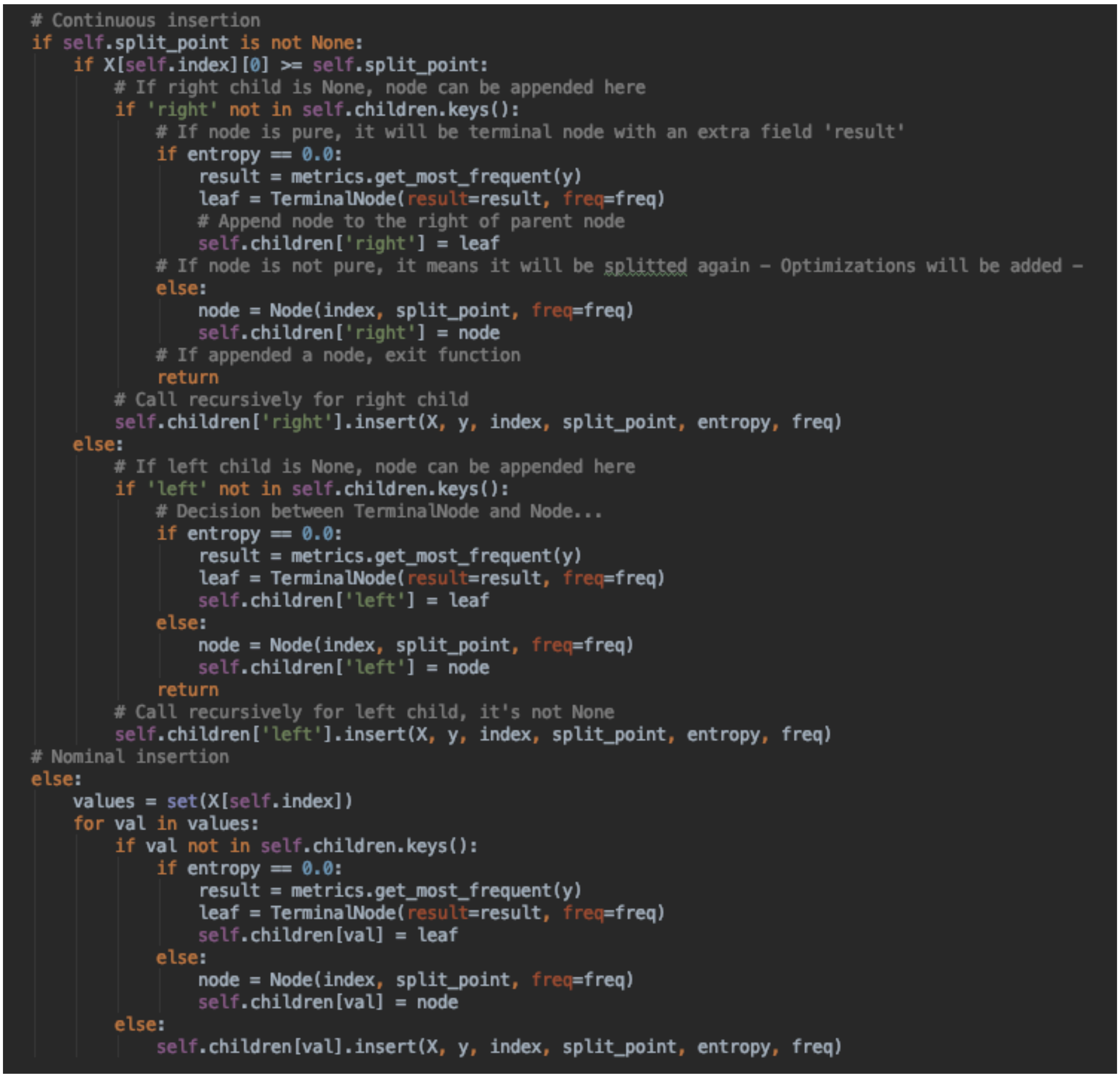
Here is the attributes of class Node.



```python
class Node:
    def __init__(self, index, split_point, freq):
        """
        :param index: index of referencing column
        :param split_point: where to split
        :param freq: frequencies of sample targets, we are trying to make them pure, but prevent overfitting

        """
        self.index = index
        self.split_point = split_point
        self.freq = freq
        # Children nodes
        self.children = {}
```

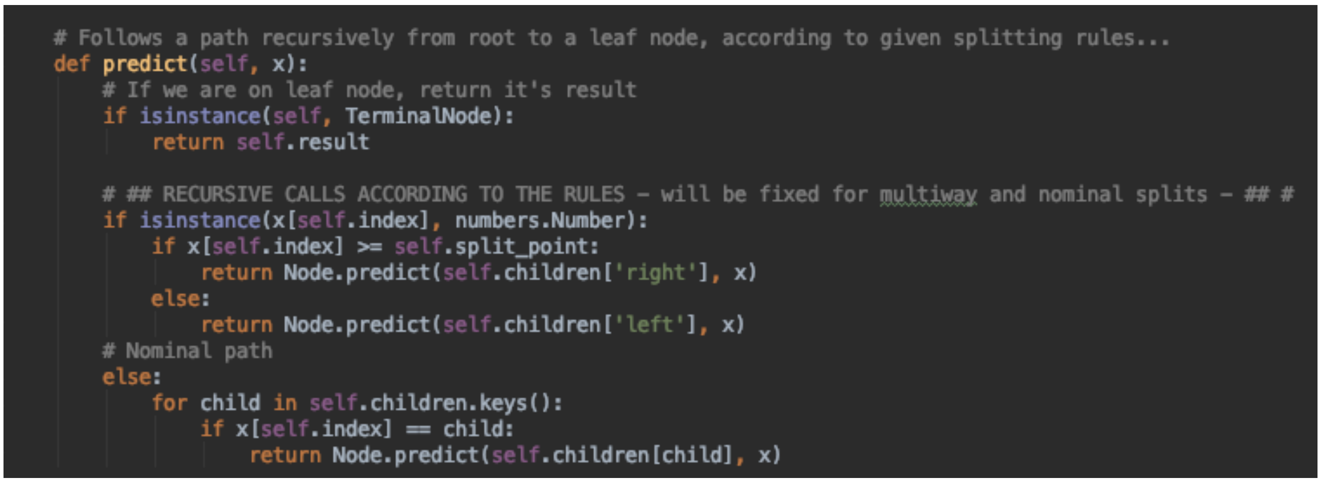
Class Node mainly stands for decision node. Contains referencing column(index), if it's a numerical node split_point is the split point chosen; if it's a categorical node, split_point is None. Variable freq contains target frequencies of the sample data which created the Node object. Variable children has references to the children nodes, it's type is dictionary, so its keys will be named 'right', 'left' for continuous data; 'right' for >=, 'left' for <. They'll be named with distinct category names for categorical data.

One of the most important methods of the class Node is the insertion method. There are two different traversing possibilities; one is passing through continuous node, other one is passing through nominal node. Implementation goes below:



```
# Continuous insertion
if self.split_point is not None:
    if X[self.index][0] >= self.split_point:
        # If right child is None, node can be appended here
        if 'right' not in self.children.keys():
            # If node is pure, it will be terminal node with an extra field 'result'
            if entropy == 0.0:
                result = metrics.get_most_frequent(y)
                leaf = TerminalNode(result=result, freq=freq)
                # Append node to the right of parent node
                self.children['right'] = leaf
            # If node is not pure, it means it will be splitted again - Optimizations will be added -
            else:
                node = Node(index, split_point, freq=freq)
                self.children['right'] = node
            # If appended a node, exit function
            return
        # Call recursively for right child
        self.children['right'].insert(X, y, index, split_point, entropy, freq)
    else:
        # If left child is None, node can be appended here
        if 'left' not in self.children.keys():
            # Decision between TerminalNode and Node...
            if entropy == 0.0:
                result = metrics.get_most_frequent(y)
                leaf = TerminalNode(result=result, freq=freq)
                self.children['left'] = leaf
            else:
                node = Node(index, split_point, freq=freq)
                self.children['left'] = node
            return
        # Call recursively for left child, it's not None
        self.children['left'].insert(X, y, index, split_point, entropy, freq)
# Nominal insertion
else:
    values = set(X[self.index])
    for val in values:
        if val not in self.children.keys():
            if entropy == 0.0:
                result = metrics.get_most_frequent(y)
                leaf = TerminalNode(result=result, freq=freq)
                self.children[val] = leaf
            else:
                node = Node(index, split_point, freq=freq)
                self.children[val] = node
        else:
            self.children[val].insert(X, y, index, split_point, entropy, freq)
```

There is also an important method, predict, which is this:

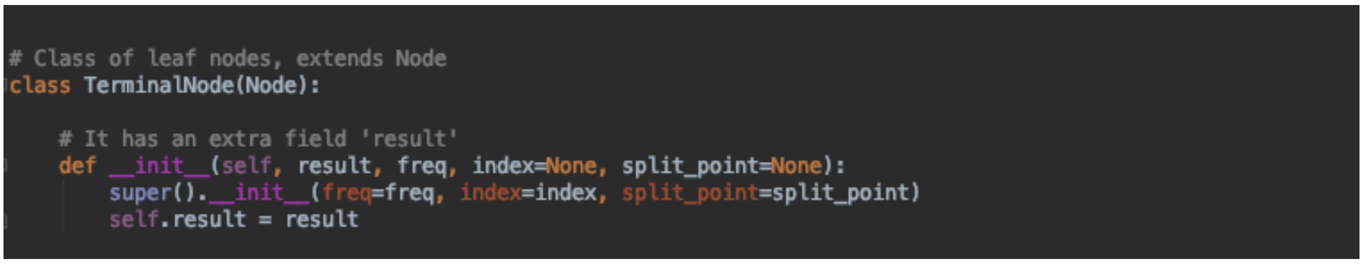


```
# Follows a path recursively from root to a leaf node, according to given splitting rules...
def predict(self, x):
    # If we are on leaf node, return it's result
    if isinstance(self, TerminalNode):
        return self.result

    # ## RECURSIVE CALLS ACCORDING TO THE RULES - will be fixed for multiway and nominal splits - ## #
    if isinstance(x[self.index], numbers.Number):
        if x[self.index] >= self.split_point:
            return Node.predict(self.children['right'], x)
        else:
            return Node.predict(self.children['left'], x)
    # Nominal path
    else:
        for child in self.children.keys():
            if x[self.index] == child:
                return Node.predict(self.children[child], x)
```

Predict method is for classifying an unseen data. It traverses tree until reaching a terminal node. Here is two possibilities, passing through nominal node and passing through continuous node. This method is working because of strict branch naming rules, e.g. naming left or right according to >= or < from split point, but these rules are not valid for flexibility.

- TerminalNode

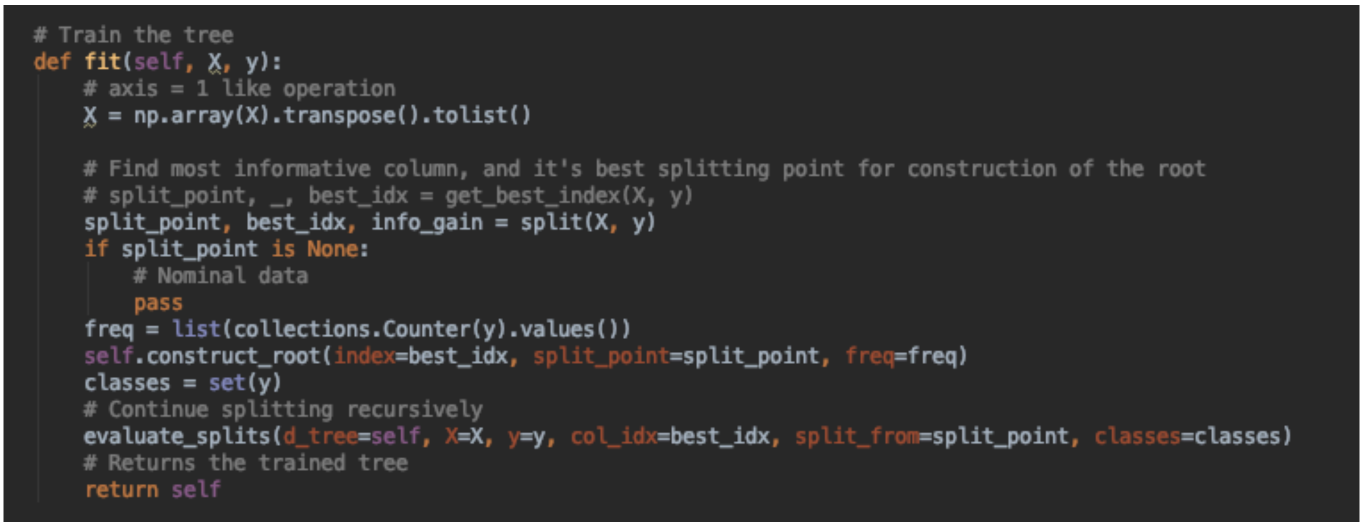
This class extends class Node. This class is for leaf nodes. It's objective is to store the classification result of the path from root to that leaf node. It has an extra field for classification result. It has no split point or referencing column, because we do not split that terminal node further. It's the end of the path.



```
# Class of leaf nodes, extends Node
class TerminalNode(Node):

    # It has an extra field 'result'
    def __init__(self, result, freq, index=None, split_point=None):
        super().__init__(freq=freq, index=index, split_point=split_point)
        self.result = result
```

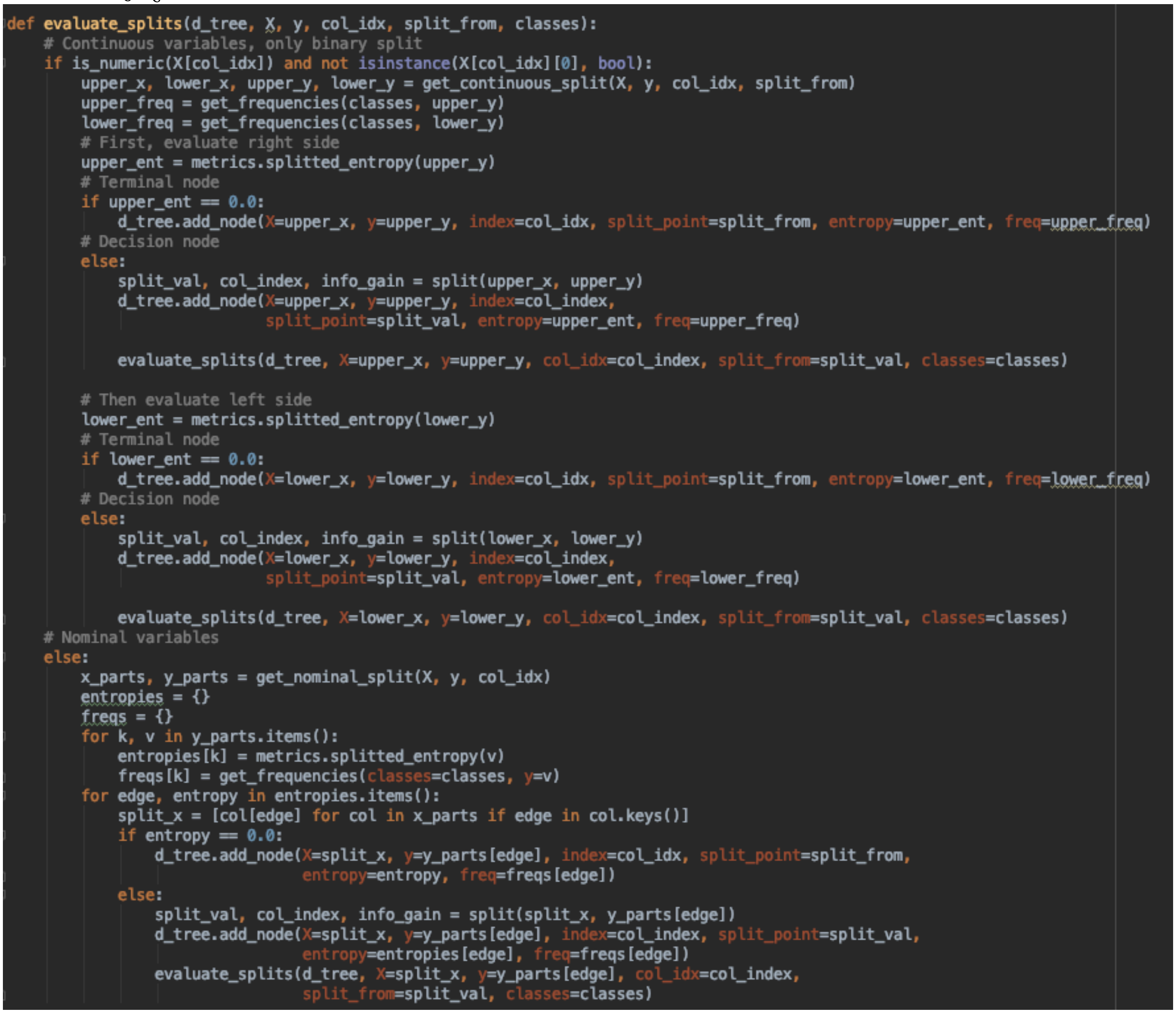
- DecisionTree

This class is for encapsulation of class Node. It contains root of the tree, which is a Node object. Its most important method is the 'fit' method. This method is for training, constructing the tree.



```
# Train the tree
def fit(self, X, y):
    # axis = 1 like operation
    X = np.array(X).transpose().tolist()

    # Find most informative column, and it's best splitting point for construction of the root
    # split_point, _, best_idx = get_best_index(X, y)
    split_point, best_idx, info_gain = split(X, y)
    if split_point is None:
        # Nominal data
        pass
    freq = list(collections.Counter(y).values())
    self.construct_root(index=best_idx, split_point=split_point, freq=freq)
    classes = set(y)
    # Continue splitting recursively
    evaluate_splits(d_tree=self, X=X, y=y, col_idx=best_idx, split_from=split_point, classes=classes)
    # Returns the trained tree
    return self
```

X parameter(independent variables) expected 2 dimensional list/array, y parameter(target variable) expected 1 dimensional array. First, we find root node, that means find the most informative split. After finding most informative split, pass its information to the class Node's constructor. Now we have root. Then call recursive evaluate_splits function. This function is very important, because recursive splits done here. This function will be discussed on the next page.



```python
def evaluate_splits(d_tree, X, y, col_idx, split_from, classes):
    # Continuous variables, only binary split
    if is_numeric(X[col_idx]) and not isinstance(X[col_idx][0], bool):
        upper_x, lower_x, upper_y, lower_y = get_continuous_split(X, y, col_idx, split_from)
        upper_freq = get_frequencies(classes, upper_y)
        lower_freq = get_frequencies(classes, lower_y)
        # First, evaluate right side
        upper_ent = metrics.splitted_entropy(upper_y)
        # Terminal node
        if upper_ent == 0.0:
            d_tree.add_node(X=upper_x, y=upper_y, index=col_idx, split_point=split_from, entropy=upper_ent, freq=upper_freq)
        # Decision node
        else:
            split_val, col_index, info_gain = split(upper_x, upper_y)
            d_tree.add_node(X=upper_x, y=upper_y, index=col_index,
                            split_point=split_val, entropy=upper_ent, freq=upper_freq)

            evaluate_splits(d_tree, X=upper_x, y=upper_y, col_idx=col_index, split_from=split_val, classes=classes)

        # Then evaluate left side
        lower_ent = metrics.splitted_entropy(lower_y)
        # Terminal node
        if lower_ent == 0.0:
            d_tree.add_node(X=lower_x, y=lower_y, index=col_idx, split_point=split_from, entropy=lower_ent, freq=lower_freq)
        # Decision node
        else:
            split_val, col_index, info_gain = split(lower_x, lower_y)
            d_tree.add_node(X=lower_x, y=lower_y, index=col_index,
                            split_point=split_val, entropy=lower_ent, freq=lower_freq)

            evaluate_splits(d_tree, X=lower_x, y=lower_y, col_idx=col_index, split_from=split_val, classes=classes)
    # Nominal variables
    else:
        x_parts, y_parts = get_nominal_split(X, y, col_idx)
        entropies = {}
        freqs = {}
        for k, v in y_parts.items():
            entropies[k] = metrics.splitted_entropy(v)
            freqs[k] = get_frequencies(classes=classes, y=v)
        for edge, entropy in entropies.items():
            split_x = [col[edge] for col in x_parts if edge in col.keys()]
            if entropy == 0.0:
                d_tree.add_node(X=split_x, y=y_parts[edge], index=col_idx, split_point=split_from,
                                entropy=entropy, freq=freqs[edge])
            else:
                split_val, col_index, info_gain = split(split_x, y_parts[edge])
                d_tree.add_node(X=split_x, y=y_parts[edge], index=col_index, split_point=split_val,
                                entropy=entropies[edge], freq=freqs[edge])
                evaluate_splits(d_tree, X=split_x, y=y_parts[edge], col_idx=col_index,
                                split_from=split_val, classes=classes)
```

For continuous splits, upper stands for the data >= split point, lower stands for the data < split point. After splitting the data, this function checks is any part pure or not. If pure(entropy is 0), this pure data appends a TerminalNode object to a valid position.

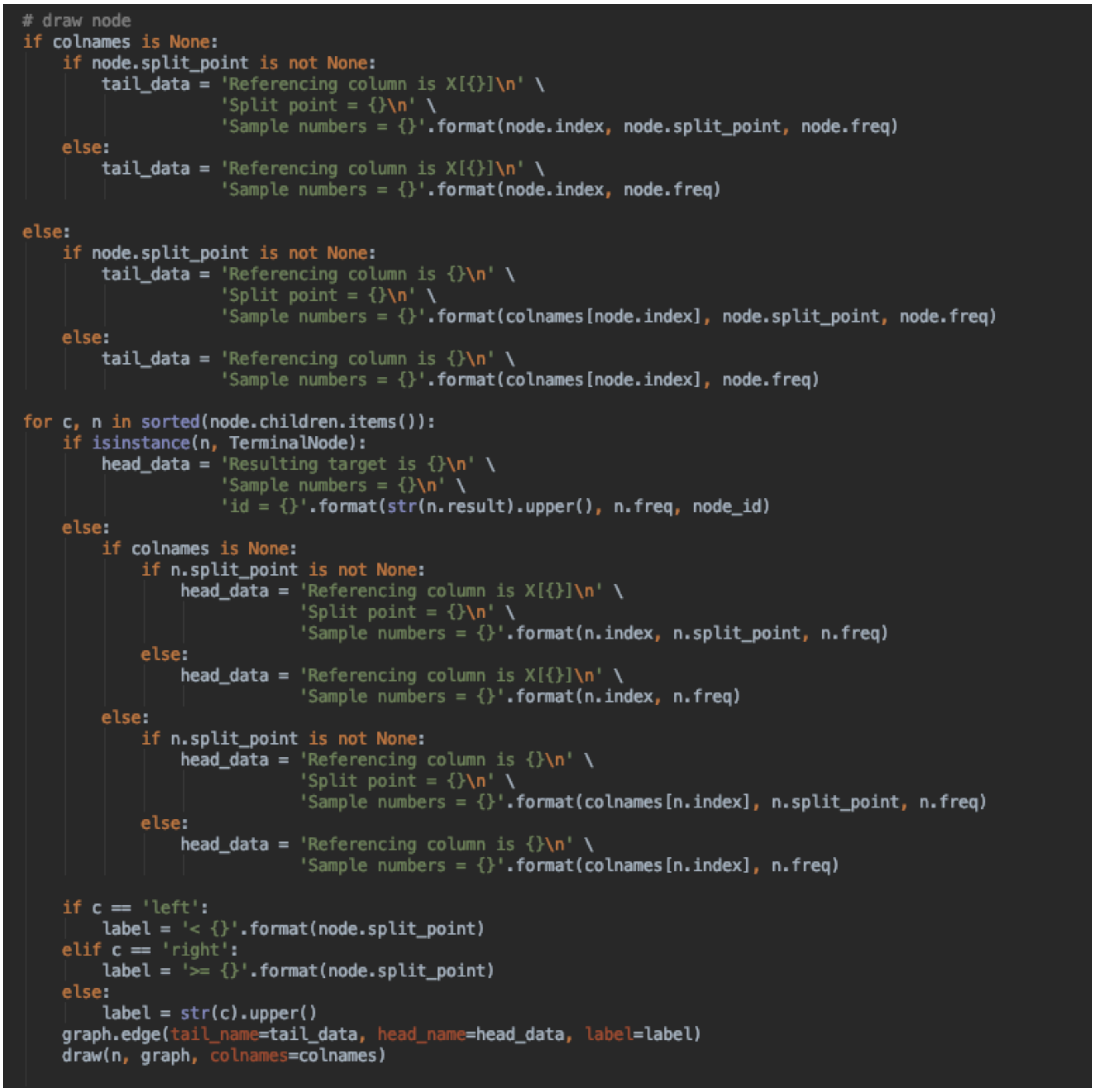
If not pure, a Node(normal decision node, not TerminalNode) object should be created, but that split data must be split again. Maybe it will be split from another column, following recursive calls decide which column to split on.

In conclusion, this function splits the data, create nodes for these after-split data until they become pure(by way of recursive calls with split data), that means they become terminal nodes. Although this operation causes overfitting, it's a good start for recursive decision tree construction.

In order to avoid overfitting, I have learned that we can put limits to terminal node's minimum samples, for example we can say that do not split nodes that contain less than 10% of the total data amount, or we can put limit directly to the entropy. Here I'm creating terminal node if and only if the data is pure. I could have said that 'Create terminal node if entropy is less than 0.1'. This operation will be done later weeks.

My supervisor told me that visualization is the most important thing about a decision tree. It's the most understandable machine learning model for the non-technical people. Also a decision tree can be written as a set of rules.

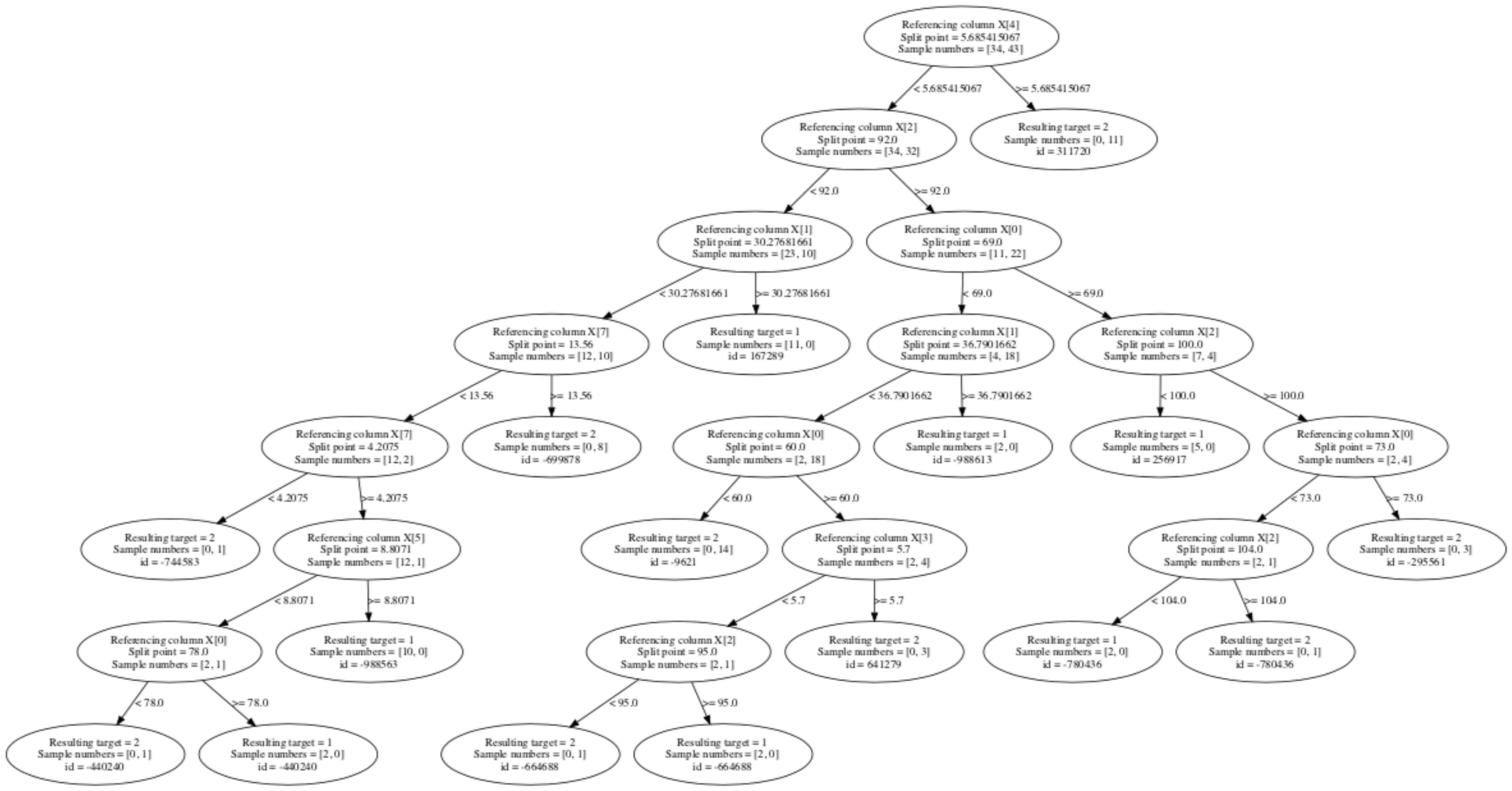
I know a method from scikit-learn's decision tree, called 'export_graphviz'. It's for visualizing the constructed decision tree. I started searching graphviz library, then I found out that I can draw any graph I want. Here is my draw function:



```
# draw node
if colnames is None:
    if node.split_point is not None:
        tail_data = 'Referencing column is X[{}]\n' \
                    'Split point = {}\n' \
                    'Sample numbers = {}'.format(node.index, node.split_point, node.freq)
    else:
        tail_data = 'Referencing column is X[{}]\n' \
                    'Sample numbers = {}'.format(node.index, node.freq)

else:
    if node.split_point is not None:
        tail_data = 'Referencing column is {}\n' \
                    'Split point = {}\n' \
                    'Sample numbers = {}'.format(colnames[node.index], node.split_point, node.freq)
    else:
        tail_data = 'Referencing column is {}\n' \
                    'Sample numbers = {}'.format(colnames[node.index], node.freq)

for c, n in sorted(node.children.items()):
    if isinstance(n, TerminalNode):
        head_data = 'Resulting target is {}\n' \
                    'Sample numbers = {}\n' \
                    'id = {}'.format(str(n.result).upper(), n.freq, node_id)
    else:
        if colnames is None:
            if n.split_point is not None:
                head_data = 'Referencing column is X[{}]\n' \
                            'Split point = {}\n' \
                            'Sample numbers = {}'.format(n.index, n.split_point, n.freq)
            else:
                head_data = 'Referencing column is X[{}]\n' \
                            'Sample numbers = {}'.format(n.index, n.freq)
        else:
            if n.split_point is not None:
                head_data = 'Referencing column is {}\n' \
                            'Split point = {}\n' \
                            'Sample numbers = {}'.format(colnames[n.index], n.split_point, n.freq)
            else:
                head_data = 'Referencing column is {}\n' \
                            'Sample numbers = {}'.format(colnames[n.index], n.freq)

    if c == 'left':
        label = '< {}'.format(node.split_point)
    elif c == 'right':
        label = '>= {}'.format(node.split_point)
    else:
        label = str(c).upper()
    graph.edge(tail_name=tail_data, head_name=head_data, label=label)
    draw(n, graph, colnames=colnames)
```

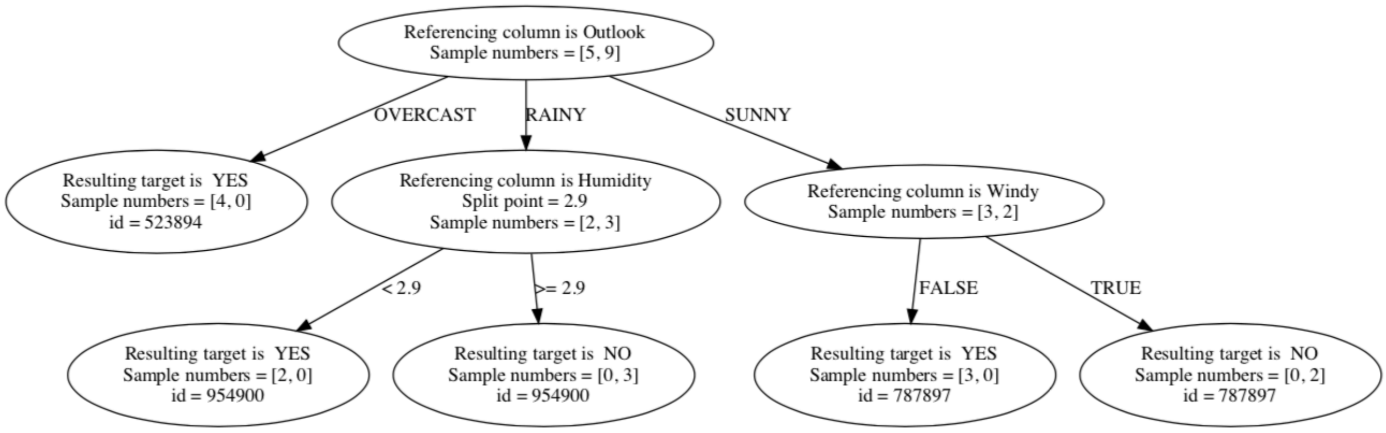
I traverse through the decision tree, get their data, draw edges from parent to the all children. Of course, draw function is recursive in order to traverse the tree.

Some examples from my first visualizations:


Referencing column X[4] Split point = 5.685415067 Sample numbers = [34, 43]
< 5.685415067
>= 5.685415067
Referencing column X[2] Split point = 92.0 Sample numbers = [34, 32]
Resulting target = 2 Sample numbers = [0, 11] id = 311720
< 92.0
>= 92.0
Referencing column X[1] Split point = 30.27681661 Sample numbers = [23, 10]
Referencing column X[0] Split point = 69.0 Sample numbers = [11, 22]
< 30.27681661
>= 30.27681661
< 69.0
>= 69.0
Referencing column X[7] Split point = 13.56 Sample numbers = [12, 10]
Resulting target = 1 Sample numbers = [11, 0] id = 167289
Referencing column X[1] Split point = 36.7901662 Sample numbers = [4, 18]
Referencing column X[2] Split point = 100.0 Sample numbers = [7, 4]
< 13.56
>= 13.56
< 36.7901662
>= 36.7901662
< 100.0
>= 100.0
Referencing column X[7] Split point = 4.2075 Sample numbers = [12, 2]
Resulting target = 2 Sample numbers = [0, 8] id = -699878
Referencing column X[0] Split point = 60.0 Sample numbers = [2, 18]
Resulting target = 1 Sample numbers = [2, 0] id = -988613
Resulting target = 1 Sample numbers = [5, 0] id = 256917
Referencing column X[0] Split point = 73.0 Sample numbers = [2, 4]
< 4.2075
>= 4.2075
< 60.0
>= 60.0
< 73.0
>= 73.0
Resulting target = 2 Sample numbers = [0, 1] id = -744583
Referencing column X[5] Split point = 8.8071 Sample numbers = [12, 1]
Resulting target = 2 Sample numbers = [0, 14] id = -9621
Referencing column X[3] Split point = 5.7 Sample numbers = [2, 4]
Referencing column X[2] Split point = 104.0 Sample numbers = [2, 1]
Resulting target = 2 Sample numbers = [0, 3] id = -295561
< 8.8071
>= 8.8071
< 5.7
>= 5.7
< 104.0
>= 104.0
Referencing column X[0] Split point = 78.0 Sample numbers = [2, 1]
Resulting target = 1 Sample numbers = [10, 0] id = -988563
Referencing column X[2] Split point = 95.0 Sample numbers = [2, 1]
Resulting target = 2 Sample numbers = [0, 3] id = 641279
Resulting target = 1 Sample numbers = [2, 0] id = -780436
Resulting target = 2 Sample numbers = [0, 1] id = -780436
< 78.0
>= 78.0
< 95.0
>= 95.0
Resulting target = 2 Sample numbers = [0, 1] id = -440240
Resulting target = 1 Sample numbers = [2, 0] id = -440240
Resulting target = 2 Sample numbers = [0, 1] id = -664688
Resulting target = 1 Sample numbers = [2, 0] id = -664688

Here is fully overfit decision tree as expected. Terminal nodes contains very little samples.

This tree is for tennis dataset, I gave it as an example at Page 7. I only changed Humidity column on the dataset with numerical values in order to see that my tree can handle both continuous and discrete data.


Referencing column is Outlook Sample numbers = [5, 9]
OVERCAST
RAINY
SUNNY
Resulting target is YES Sample numbers = [4, 0] id = 523894
Referencing column is Humidity Split point = 2.9 Sample numbers = [2, 3]
Referencing column is Windy Sample numbers = [3, 2]
< 2.9
>= 2.9
FALSE
TRUE
Resulting target is YES Sample numbers = [2, 0] id = 954900
Resulting target is NO Sample numbers = [0, 3] id = 954900
Resulting target is YES Sample numbers = [3, 0] id = 787897
Resulting target is NO Sample numbers = [0, 2] id = 787897

As we can see, it looks like it’s successful at handling both continuous and discrete values.

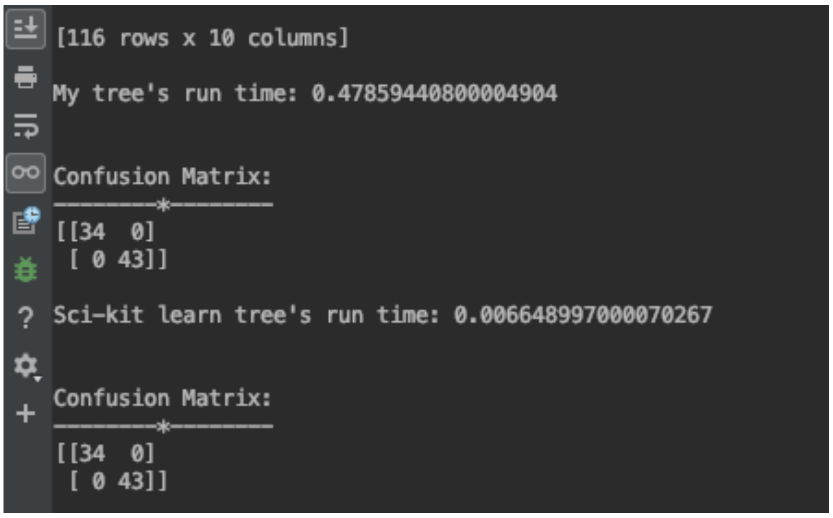
c. Week 3

My problem at this week is that my decision tree works too slow for relatively big datasets. For example, processing a dataset containing 500 observations, 9 columns took almost 2 minutes. When I asked my supervisor 'How many rows and columns do you have in the data you are using?', he said 'At least one million observations, almost one hundred columns we have'. Then I realized that it's impossible to work on real word data with my decision tree implementation.

I started looking for a solution to this problem. I know that what makes a program slow generally. It's mostly caused by input size dependent loops. I used Python lists and Python for loops almost everywhere, while storing the data, copying the data etc. Then I learned that Python lists are very very slow. While working with big data, NumPy arrays should be used.

NumPy arrays are a little bit different from Python lists. Python lists are heterogenous, however NumPy arrays are homogenous like C arrays. If we want to use NumPy arrays heterogenous, we can pass the datatype as 'object', like Object arrays in Java. Advantage of NumPy arrays is that it focuses on handle jobs without loops. If loops must be used, NumPy library implements these loops in C language or in Cython, which are way faster than Python. I saw that some array operations in NumPy are x40 faster than native Python operations.

In the third week of my training, I started to convert every Python lists I use to NumPy arrays. I also wanted to give better look to my visualization and get a better object oriented design.



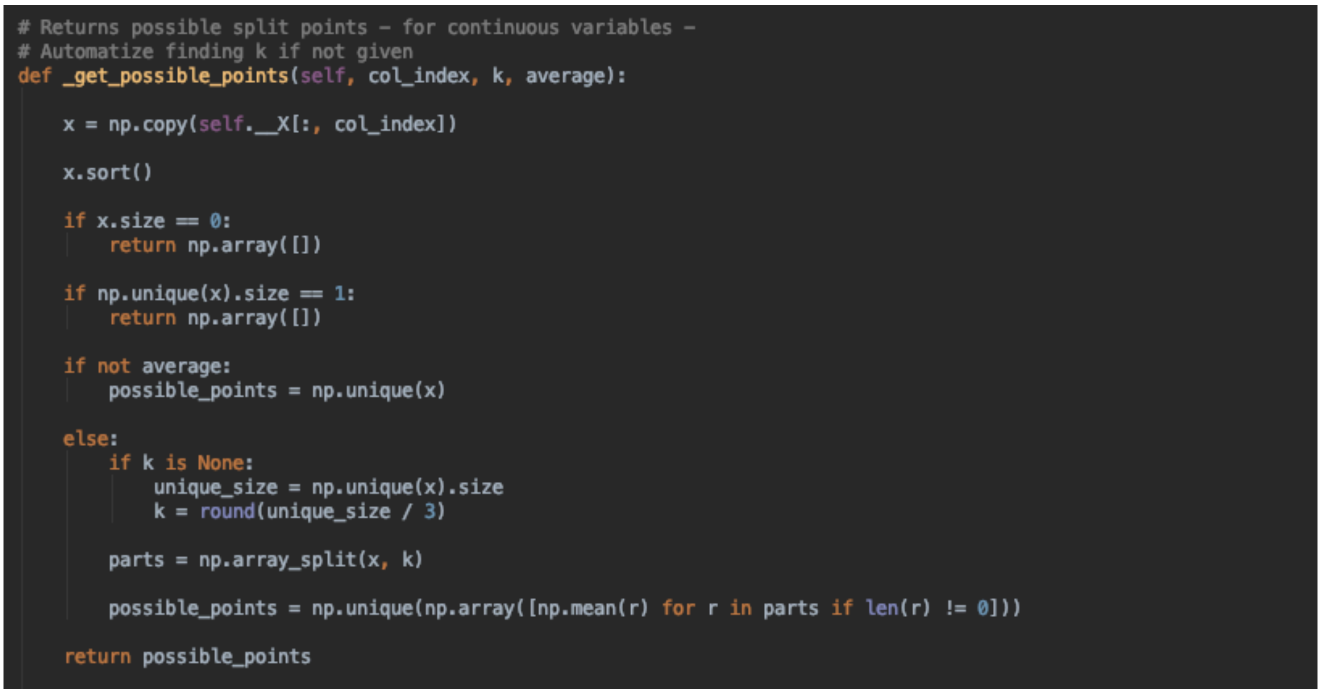
```
[116 rows x 10 columns]

My tree's run time: 0.47859440800004904


Confusion Matrix:
--------*--------
[[34  0]
 [ 0 43]]

Sci-kit learn tree's run time: 0.006648997000070267


Confusion Matrix:
--------*--------
[[34  0]
 [ 0 43]]
```

Here we can see, there is a lot execution time difference between scikit-learn's tree training and my tree training with a dataset sized 116 x 10.

Also my current visualization is not looking well. Most of the decision tree visualizations are colored according to their odds ratio.

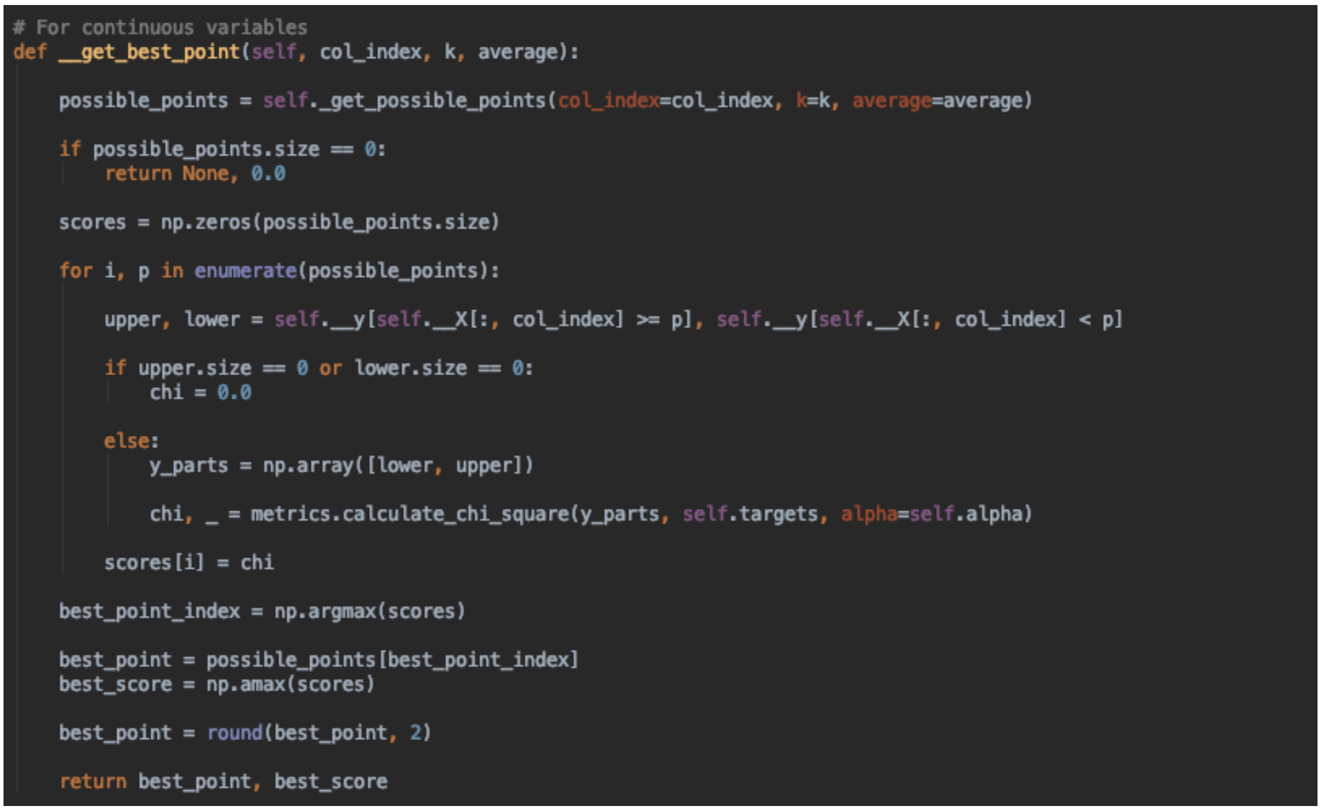
For speeding up the program, I firstly decide to change my split point selection method. I was trying every possible value, then choose the most informative split point. Most probably this selection method is the thing that slows the program.

After talking with my supervisor, I started to test a new discretization method. First sort the values in the column, then take its distinct values(maybe we can get smaller set by this way). After that, chunk these possible points to k parts, k is user specified, take each part’s mean. These means are the possible split points. Test all of them, get the most informative split point.



```python
# Returns possible split points - for continuous variables -
# Automatize finding k if not given
def _get_possible_points(self, col_index, k, average):

    x = np.copy(self.__X[:, col_index])

    x.sort()

    if x.size == 0:
        return np.array([])

    if np.unique(x).size == 1:
        return np.array([])

    if not average:
        possible_points = np.unique(x)

    else:
        if k is None:
            unique_size = np.unique(x).size
            k = round(unique_size / 3)

        parts = np.array_split(x, k)

        possible_points = np.unique(np.array([np.mean(r) for r in parts if len(r) != 0]))

    return possible_points
```

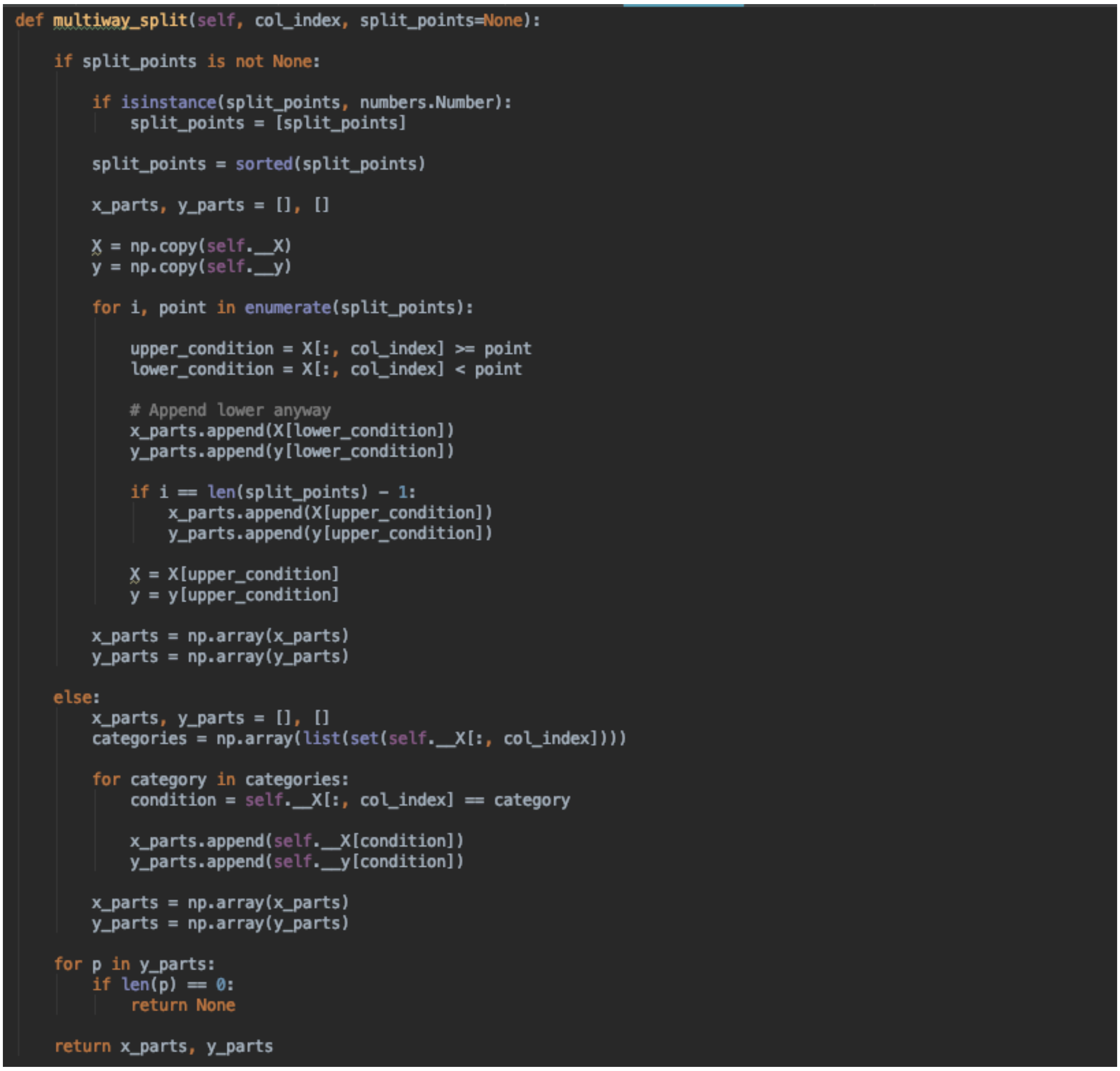
Here is the code. ’np’ here stands for NumPy library. As we can see, most of the methods I use are from NumPy library.



```python
# For continuous variables
def __get_best_point(self, col_index, k, average):

    possible_points = self._get_possible_points(col_index=col_index, k=k, average=average)

    if possible_points.size == 0:
        return None, 0.0

    scores = np.zeros(possible_points.size)

    for i, p in enumerate(possible_points):

        upper, lower = self.__y[self.__X[:, col_index] >= p], self.__y[self.__X[:, col_index] < p]

        if upper.size == 0 or lower.size == 0:
            chi = 0.0

        else:
            y_parts = np.array([lower, upper])

            chi, _ = metrics.calculate_chi_square(y_parts, self.targets, alpha=self.alpha)

        scores[i] = chi

    best_point_index = np.argmax(scores)

    best_point = possible_points[best_point_index]
    best_score = np.amax(scores)

    best_point = round(best_point, 2)

    return best_point, best_score
```

Second code piece on the previous page tests all possible split points. Measure their quality of split for each split point with specified metric, then returns the best possible split point and its metric score. Again lots of Python lists converted to NumPy arrays in order to speed up the program.

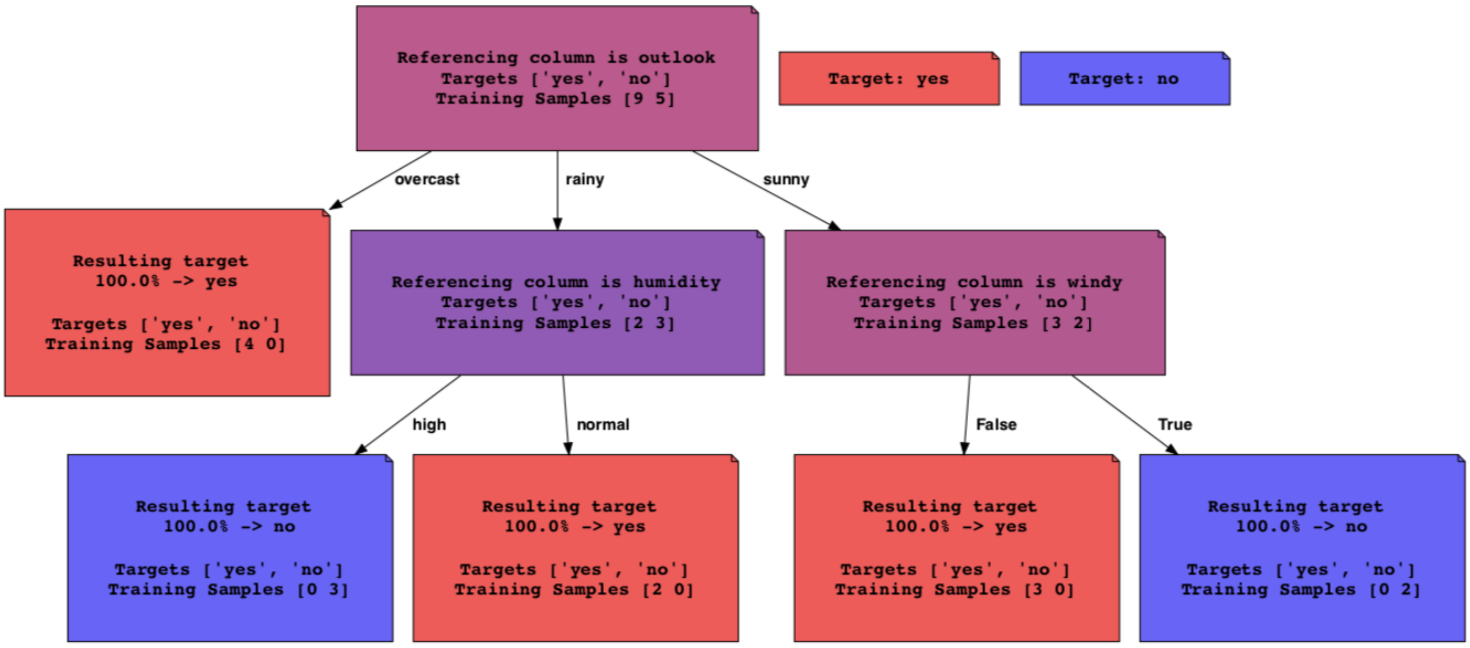
Also here is another example of converted NumPy arrays:



```python
def multiway_split(self, col_index, split_points=None):

    if split_points is not None:

        if isinstance(split_points, numbers.Number):
            split_points = [split_points]

        split_points = sorted(split_points)

        x_parts, y_parts = [], []

        X = np.copy(self.__X)
        y = np.copy(self.__y)

        for i, point in enumerate(split_points):

            upper_condition = X[:, col_index] >= point
            lower_condition = X[:, col_index] < point

            # Append lower anyway
            x_parts.append(X[lower_condition])
            y_parts.append(y[lower_condition])

            if i == len(split_points) - 1:
                x_parts.append(X[upper_condition])
                y_parts.append(y[upper_condition])

            X = X[upper_condition]
            y = y[upper_condition]

        x_parts = np.array(x_parts)
        y_parts = np.array(y_parts)

    else:
        x_parts, y_parts = [], []
        categories = np.array(list(set(self.__X[:, col_index])))

        for category in categories:
            condition = self.__X[:, col_index] == category

            x_parts.append(self.__X[condition])
            y_parts.append(self.__y[condition])

        x_parts = np.array(x_parts)
        y_parts = np.array(y_parts)

    for p in y_parts:
        if len(p) == 0:
            return None

    return x_parts, y_parts
```

Previously I used Python lists to split the data, here I convert them to the NumPy arrays. Indexing operations are faster in that case.

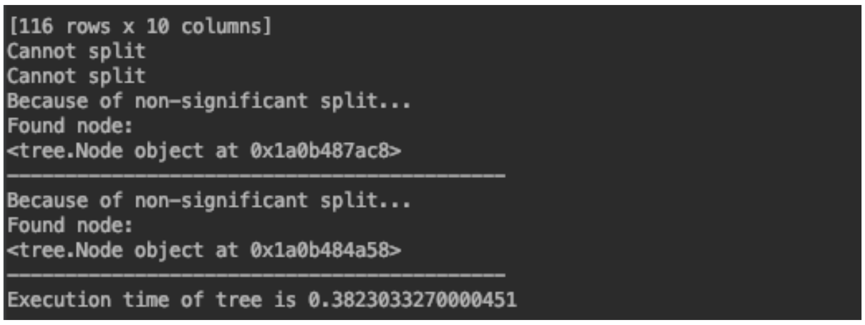
Below, my new visualization:



Pure 'yes' targets colored red, 'no' targets colored blue. These colors are mixed according to nodes' odds ratios.

I mostly worked on relatively bigger and more complex(contains both discrete and continuous variables) datasets in my training, but their visualizations are too big to put here e.g. a tree with 10 depth levels. I use this dataset for my report because it's easy to understand, easy to explain and easy to visualize.

Here is the result of speeding up the process.


```
[116 rows x 10 columns]
Cannot split
Cannot split
Because of non-significant split...
Found node:
<tree.Node object at 0x1a0b487ac8>
--------------------------------------------
Because of non-significant split...
Found node:
<tree.Node object at 0x1a0b484a58>
--------------------------------------------
Execution time of tree is 0.3823033270000451
```

After my trials to speed up the program, execution time decreased very little for this dataset. Larger decreases can be observed for bigger datasets, because algorithm has exponentially growing execution time according to the input size.

Other messages are caused by pre and post pruning, I put a limit(user specified percentage) to the data splitting condition. I also put a sample limit for the terminal nodes e.g. if a node has < n samples, this node becomes a terminal node. I also removed splits that are not significant, it means if a split is not informative so much(there is a limit, too), program removes this split.

d. Week 4

As I mentioned before, my tree handles continuous data with binary splits. That means, I only pick one split point to discretize the data. My main objective is to pick more than one split point on my last week.

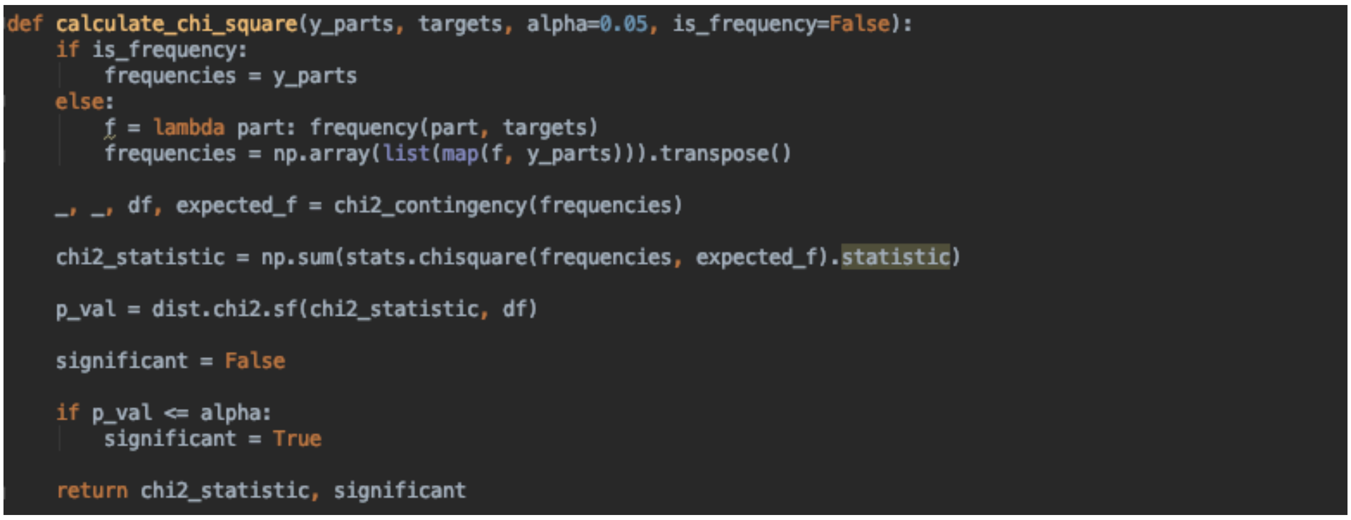
Because of my main objective is to split multiway, I needed another metric more reliable than entropy-information gain. When we are splitting multiway, we need to handle degrees of freedom. So, I started to use chi square test metric. Here is the formula:

$$\chi_c^2 = \sum \frac{(O_i - E_i)^2}{E_i}$$

O_i stands for observed frequencies of class i.

E_i stands for expected frequencies of class i.

Here is my implementation:

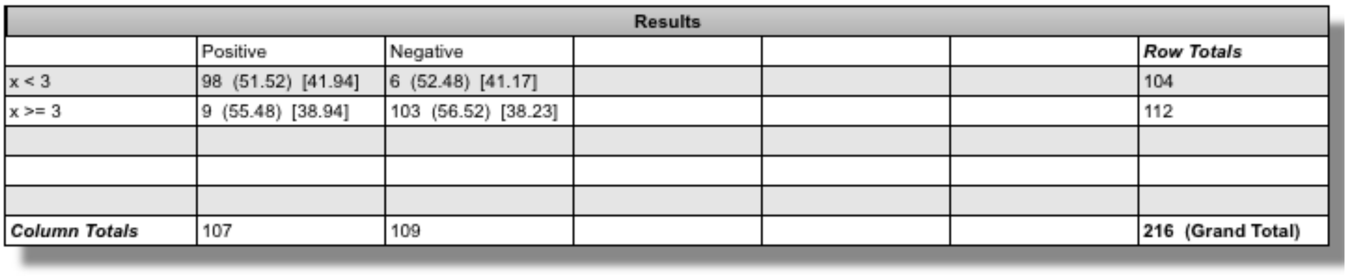


```python
def calculate_chi_square(y_parts, targets, alpha=0.05, is_frequency=False):
    if is_frequency:
        frequencies = y_parts
    else:
        f = lambda part: frequency(part, targets)
        frequencies = np.array(list(map(f, y_parts))).transpose()

    _, _, df, expected_f = chi2_contingency(frequencies)

    chi2_statistic = np.sum(stats.chisquare(frequencies, expected_f).statistic)

    p_val = dist.chi2.sf(chi2_statistic, df)

    significant = False

    if p_val <= alpha:
        significant = True

    return chi2_statistic, significant
```

Pearson's chi-squared test (χ^2) is a statistical test applied to sets of categorical data to evaluate how likely it is that any observed difference between the sets arose by chance. It's suitable for unpaired data from large samples.

Alpha is the significance level, which can be user specified. Higher chi square statistics are better, and higher chi square statistics means lower p values. Alpha is the limit for the p value, in my case it's a limit for any split's significancy, lower alpha values, e.g. alpha=0.01, can lead more significant splits.

I used some web applications to test chi square test's reliability. One example is this:



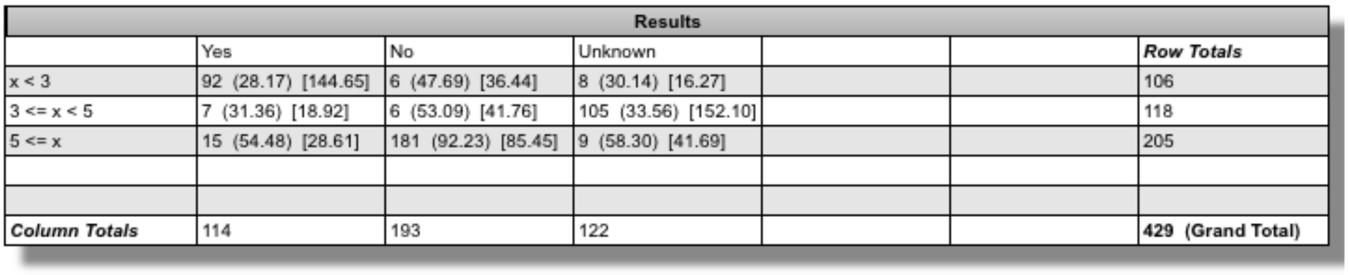
Results

	Positive	Negative				*Row Totals*
x < 3	98 (51.52) [41.94]	6 (52.48) [41.17]				104
x >= 3	9 (55.48) [38.94]	103 (56.52) [38.23]				112
Column Totals	107	109				**216** (Grand Total)

The chi-square statistic is 160.2727. The *p*-value is < .00001. The result is significant at *p* < .05.

For example if we have a split like this, it would be perfect for us. Chi square test tells the same to us. Chi square statistic is high and p value is too low.

Let's test for a multi class problem with a multiway split:

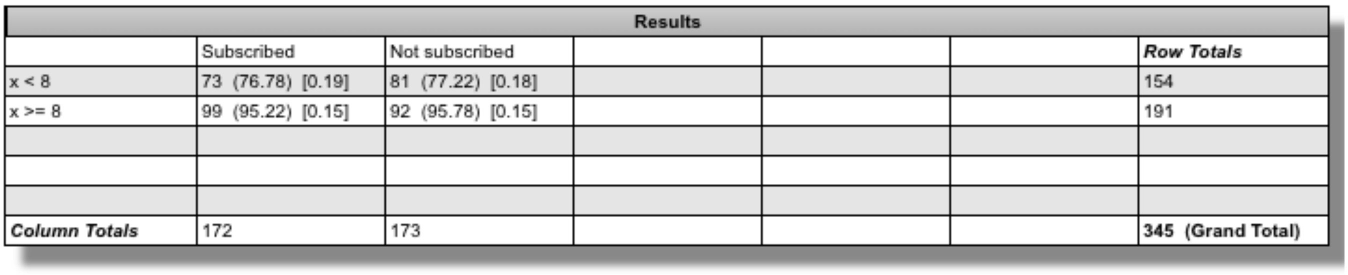


Results

	Yes	No	Unknown			*Row Totals*
x < 3	92 (28.17) [144.65]	6 (47.69) [36.44]	8 (30.14) [16.27]			106
3 <= x < 5	7 (31.36) [18.92]	6 (53.09) [41.76]	105 (33.56) [152.10]			118
5 <= x	15 (54.48) [28.61]	181 (92.23) [85.45]	9 (58.30) [41.69]			205
Column Totals	114	193	122			**429** (Grand Total)

The chi-square statistic is 565.8923. The *p*-value is < 0.00001. The result is significant at *p* < .05.

Here is one more perfect split. We can conclude split's significancy from chi square results. Now let's see a terrible split:



Results

	Subscribed	Not subscribed				*Row Totals*
x < 8	73 (76.78) [0.19]	81 (77.22) [0.18]				154
x >= 8	99 (95.22) [0.15]	92 (95.78) [0.15]				191
Column Totals	172	173				**345** (Grand Total)

The chi-square statistic is 0.6692. The *p*-value is .413318. The result is *not* significant at *p* < .05.

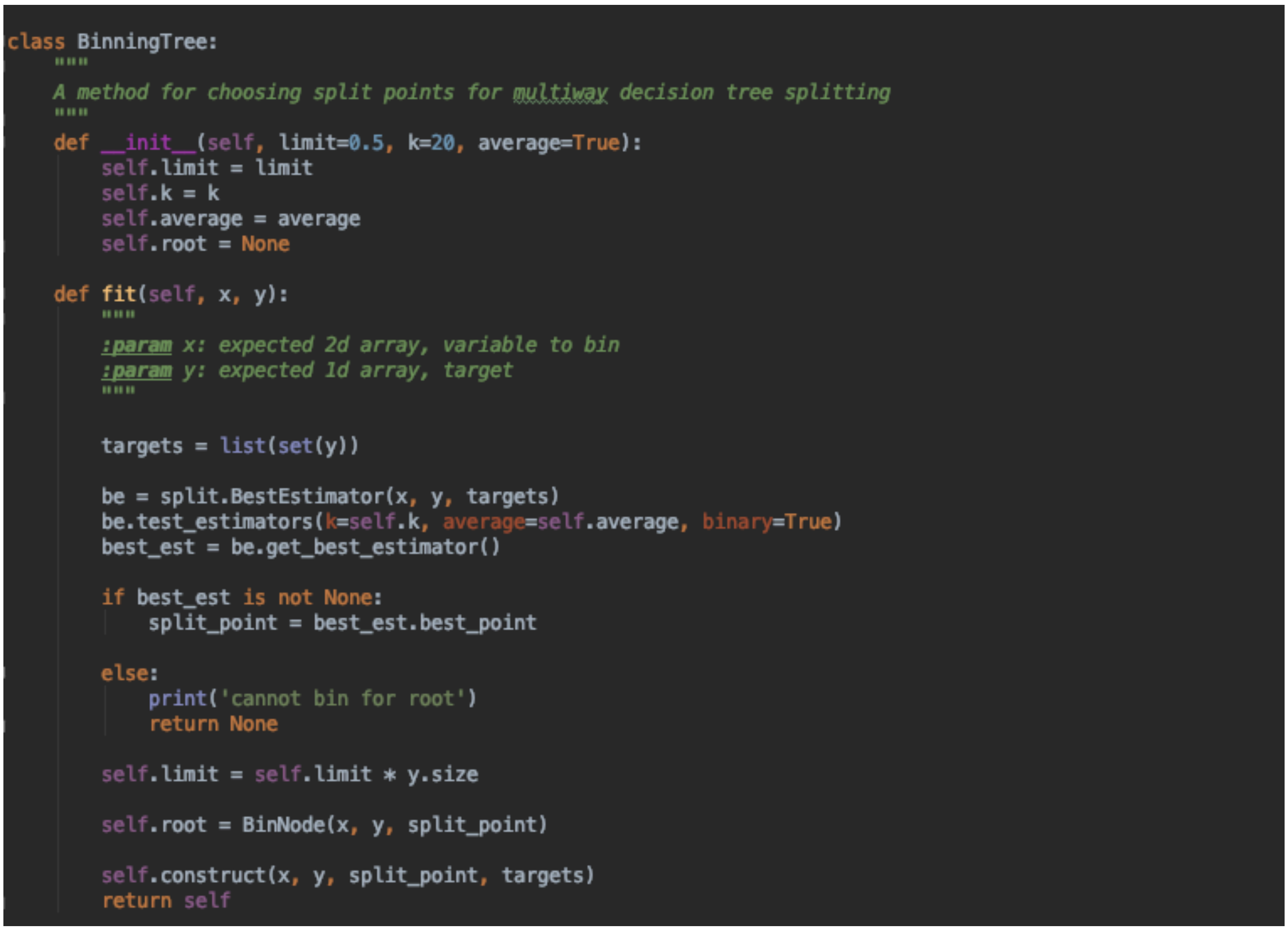
All samples are too close to each other, this is basically a terrible split. We can also see that from chi square test result.

I compared my chi square test results with this web application's results. I got the same results with this calculator. So once I verified my implementation's correctness, I started to research multiway splits, continuous data discretization.

Then I found that I can use decision trees to discretize the data. Fit the tree only with a variable that we want to discretize. This will give us best split points. If number of split points increases, complexity of the tree increases too. So I put larger limits on this tree.

I created a class BinningTree, it has no difference from a decision tree, its nodes only contains split points, because for example there is no need to store the result. We are just getting split points. Depth of this tree can be 1 at most, that means there can be 3 split points at max. Sample limit is 50% of the total data size. These parameters of course can be user specified.

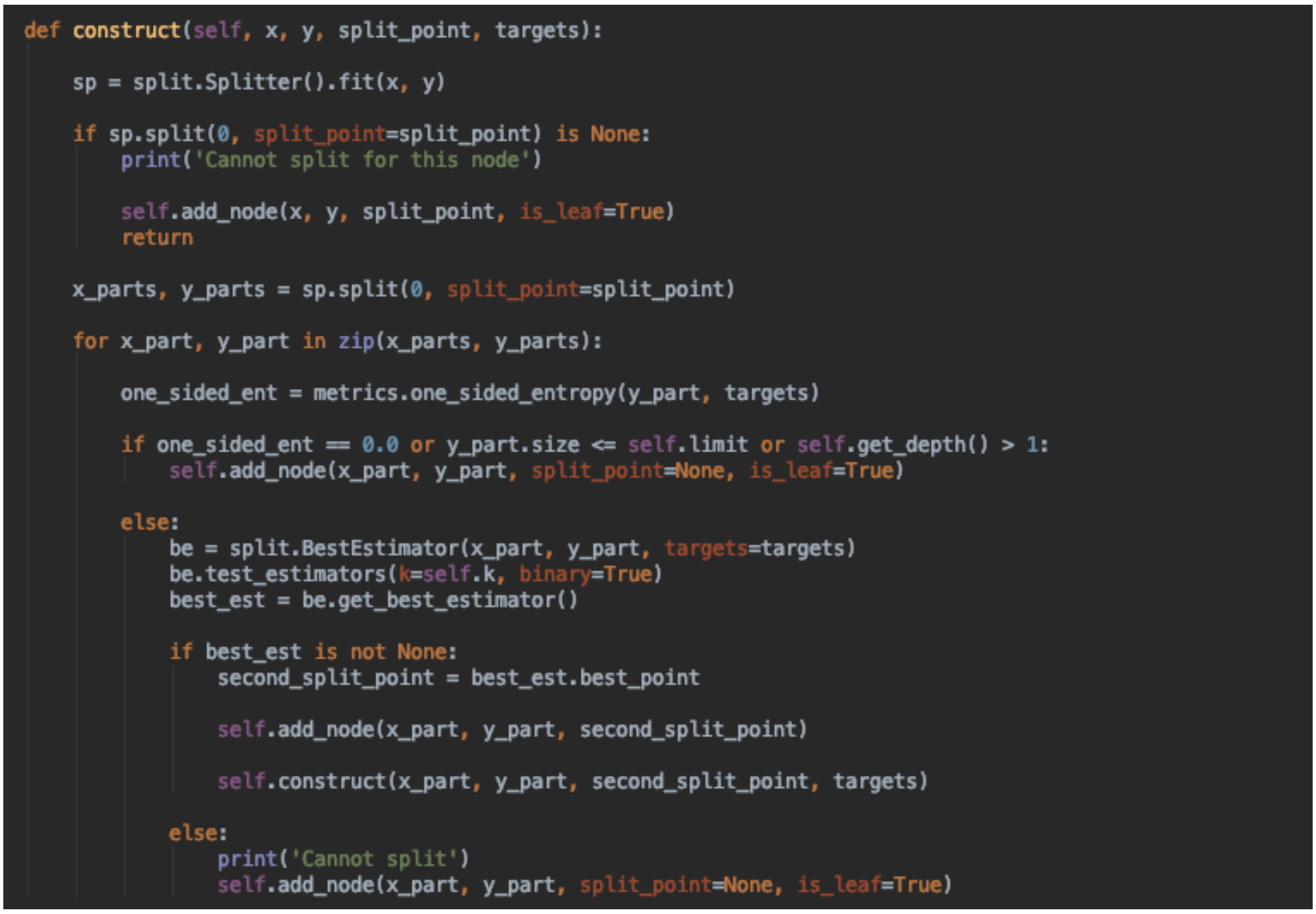
Here is my constructor and fit method. Binning tree only performs binary splits for each node.



```python
class BinningTree:
    """
    A method for choosing split points for multiway decision tree splitting
    """
    def __init__(self, limit=0.5, k=20, average=True):
        self.limit = limit
        self.k = k
        self.average = average
        self.root = None

    def fit(self, x, y):
        """
        :param x: expected 2d array, variable to bin
        :param y: expected 1d array, target
        """

        targets = list(set(y))

        be = split.BestEstimator(x, y, targets)
        be.test_estimators(k=self.k, average=self.average, binary=True)
        best_est = be.get_best_estimator()

        if best_est is not None:
            split_point = best_est.best_point

        else:
            print('cannot bin for root')
            return None

        self.limit = self.limit * y.size

        self.root = BinNode(x, y, split_point)

        self.construct(x, y, split_point, targets)
        return self
```

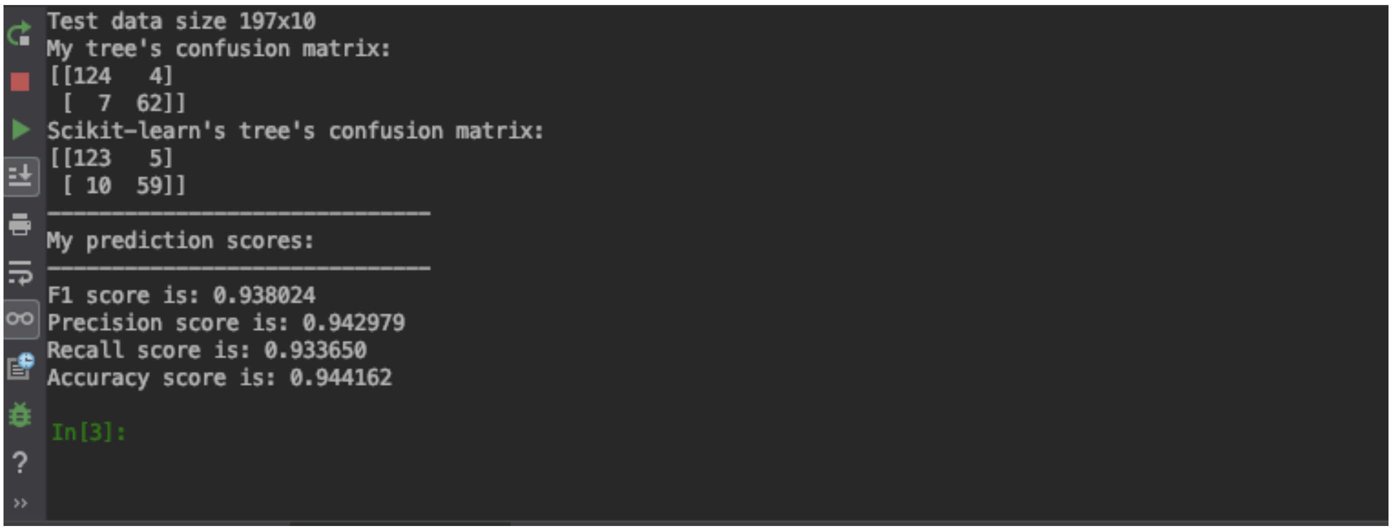
For the main decision tree, if the user doesn't want a multiway split, he/she can specify tree's characterization on continuous splits, binary or multiway.

Binning tree also have to has a recursive construction method. My implementation is on the next page. It looks like normal decision tree's construction function, however this has larger limits, also it has also a depth limit.



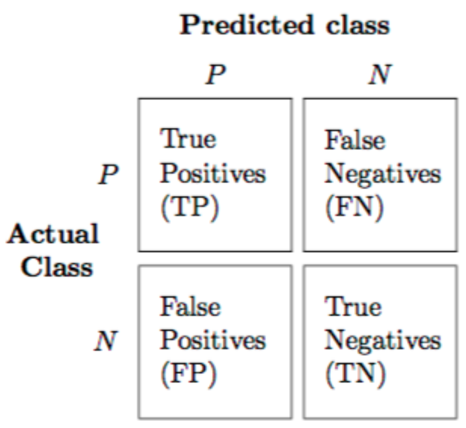
```python
def construct(self, x, y, split_point, targets):

    sp = split.Splitter().fit(x, y)

    if sp.split(0, split_point=split_point) is None:
        print('Cannot split for this node')

        self.add_node(x, y, split_point, is_leaf=True)
        return

    x_parts, y_parts = sp.split(0, split_point=split_point)

    for x_part, y_part in zip(x_parts, y_parts):

        one_sided_ent = metrics.one_sided_entropy(y_part, targets)

        if one_sided_ent == 0.0 or y_part.size <= self.limit or self.get_depth() > 1:
            self.add_node(x_part, y_part, split_point=None, is_leaf=True)

        else:
            be = split.BestEstimator(x_part, y_part, targets=targets)
            be.test_estimators(k=self.k, binary=True)
            best_est = be.get_best_estimator()

            if best_est is not None:
                second_split_point = best_est.best_point

                self.add_node(x_part, y_part, second_split_point)

                self.construct(x_part, y_part, second_split_point, targets)

            else:
                print('Cannot split')
                self.add_node(x_part, y_part, split_point=None, is_leaf=True)
```

Here is an example result of my tree’s classification performance on a breast cancer dataset. My results can be compared with scikit-learn’s tree’s results as shown in below. My tree has 4 more correct predictions than scikit-learn’s.



```
Test data size 197x10
My tree's confusion matrix:
[[124   4]
 [  7  62]]
Scikit-learn's tree's confusion matrix:
[[123   5]
 [ 10  59]]
------------------------------
My prediction scores:
------------------------------
F1 score is: 0.938024
Precision score is: 0.942979
Recall score is: 0.933650
Accuracy score is: 0.944162

In[3]:
```

- Confusion Matrix:



		Predicted class P	Predicted class N
Actual Class	P	True Positives (TP)	False Negatives (FN)
	N	False Positives (FP)	True Negatives (TN)

Confusion matrix can be used for analysis of classification performance. Main diagonal values are the samples that classified correctly.

- F1 Score

$$F_1 = \left(\frac{\mathrm{recall}^{-1} + \mathrm{precision}^{-1}}{2}\right)^{-1} = 2 \cdot \frac{\mathrm{precision} \cdot \mathrm{recall}}{\mathrm{precision} + \mathrm{recall}}.$$

It can be achieved using recall and precision scores.

- Precision Score

$$\mathrm{Precision} = \frac{tp}{tp + fp}$$

tp stands for true positives, fp stands for false positives as shown above, on the confusion matrix.

- Recall Score

$$\mathrm{Recall} = \frac{tp}{tp + fn}$$

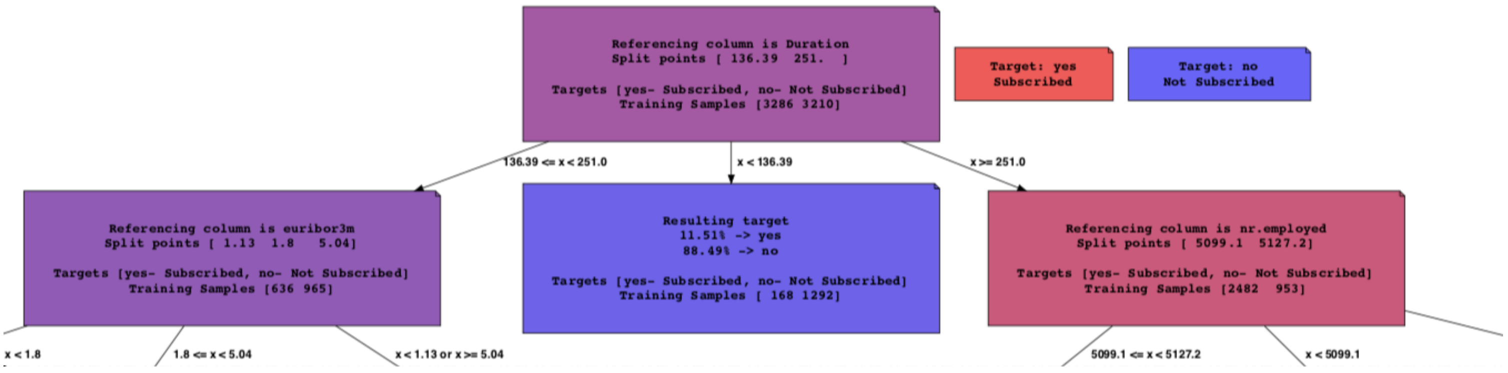
tp stands for true positives, fn stands for false negatives as shown above, on the confusion matrix.

- Accuracy Score

In the calculation of accuracy score, total correct predictions is used.

$$\mathrm{Accuracy} = \frac{tp + tn}{tp + tn + fp + fn}$$

An example of multiway split:


Referencing column is Duration
Split points [136.39 251.]
Targets [yes- Subscribed, no- Not Subscribed]
Training Samples [3286 3210]
Target: yes Subscribed
Target: no Not Subscribed
136.39 <= x < 251.0
x < 136.39
x >= 251.0
Referencing column is euribor3m
Split points [1.13 1.8 5.04]
Targets [yes- Subscribed, no- Not Subscribed]
Training Samples [636 965]
Resulting target
11.51% -> yes
88.49% -> no
Targets [yes- Subscribed, no- Not Subscribed]
Training Samples [168 1292]
Referencing column is nr.employed
Split points [5099.1 5127.2]
Targets [yes- Subscribed, no- Not Subscribed]
Training Samples [2482 953]
x < 1.8
1.8 <= x < 5.04
x < 1.13 or x >= 5.04
5099.1 <= x < 5127.2
x < 5099.1

Further splits are performed for this tree. It's too big to put it's whole picture here so I put only the root and root's children here.

5. Conclusion

This training was a great experience for me. I had chance to work with something that I'm interested in and I learned what's going on behind a decision tree. While I was implementing my tree, I used lots of useful and efficient libraries. Because I was implementing something from scratch, that leads me to learn lots of algorithms behind the scenes.

Also my supervisor answered my every question perfectly, pushing me to learn more, work more, research more about machine learning.

When I started my training, I only have interest about machine learning. When I end my training, I also have knowledge about machine learning besides interest.

6. References

- https://en.wikipedia.org/wiki/RStudio
- https://en.wikipedia.org/wiki/PyCharm
- https://en.wikipedia.org/wiki/Anaconda_(Python_distribution)
- https://en.wikipedia.org/wiki/Graphviz
- https://en.wikipedia.org/wiki/Orange_(software)
- https://en.wikipedia.org/wiki/GitHub
- http://jupyter.org/
- http://www.saedsayad.com/decision_tree.htm
- https://en.wikipedia.org/wiki/Decision_tree
- https://en.wikipedia.org/wiki/Precision_and_recall
- http://dataaspirant.com/2017/01/30/how-decision-tree-algorithm-works/
- https://www.socscistatistics.com/tests/chisquare2/Default2.aspx
- http://www.numpy.org/