Decision tree report

2015004248
Jo Hyung Jung

1. Overview

Decision tree induction is used in classification to predict categorical class labels. It has two step process.
First, model construction. By using a set of predetermined classes by using a training data, it can explain how the attributes determine the class label in training data. As it uses predetermined data, it is also called supervised learning.
After model construction, its accuracy is evaluated by using test data. If the accuracy is sufficiently high, model is ready to classify unknown data.
In this assignment, I can deeply think about implementation details to get higher accuracy.

2. Run-time environment

- OS: Ubuntu 18.04
- Language: Python 3.6
- Libraries:
  - Pandas 1.0.3: for effective access to data.
  - Python sys module: for getting command-line arguments.
  - Python math module: for calculating log value.

3. Summary of algorithm

For constructing model, we need to set classification standard. It is called attribute selection method in decision tree, as decision tree uses attribute to classify data. There are three ways to select attribute.

i) Information gain

Information gain uses the notion of ‘entropy’(Entropy denotes the heterogeneity of data set) Information gain compares the difference of entropy(Before and After splitting data set by using certain attribute) in every attributes.

ii) Gain ratio

Information gain tends to choose attribute that has more values than the others. Gain ratio uses normalization to solve this problem. It uses ‘split info’ as normalization term. Split info has similar equation form to entropy, but is uses ratio of data set size rather than probability of class label.

iii) Gini index

Gini index has totally different form from information gain and gain ratio. It assumes binary partition. New notion 'Gini' is used to represent the impurity of data set by using weighted average form. By dividing data set into two subset, it compares reduction of impurity in each subset.

After constructing model, we can evaluate test data by traversing constructed decision tree.

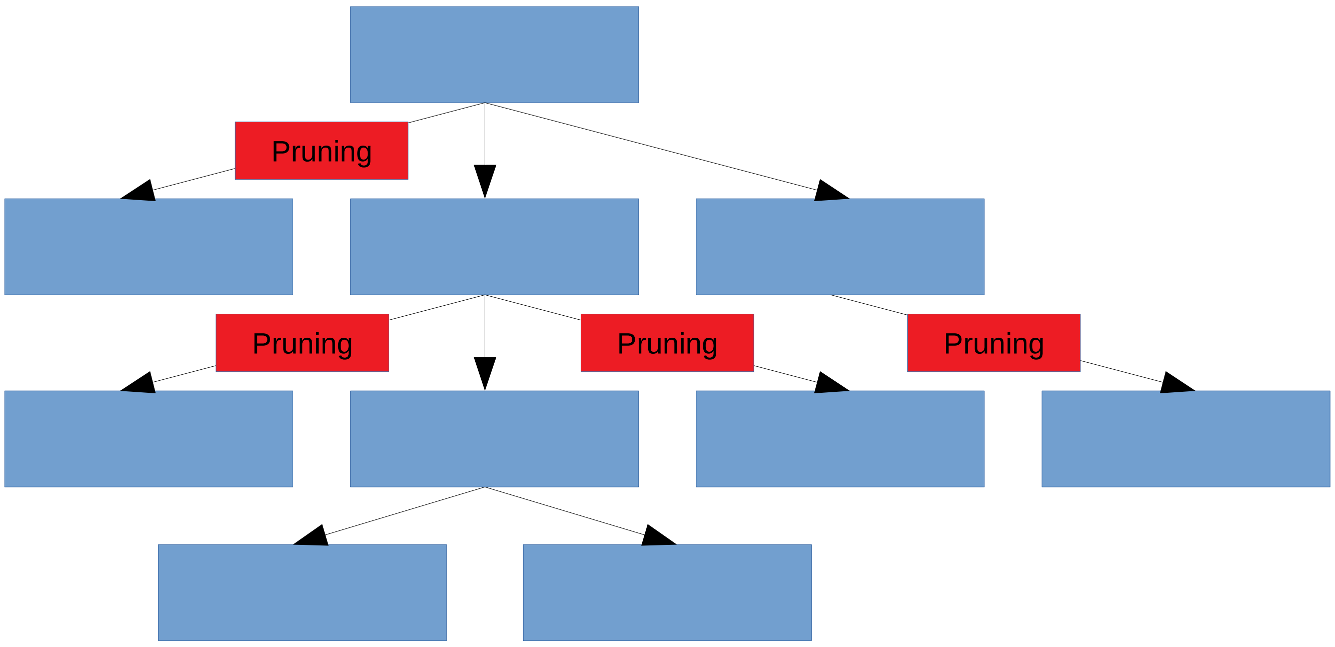
4. Implementation details

I implemented three attribute selection methods(Information gain, Gain ratio, Gini index). As this assignment is competition project, I have to add **'pruning'** feature to get higher accuracy.

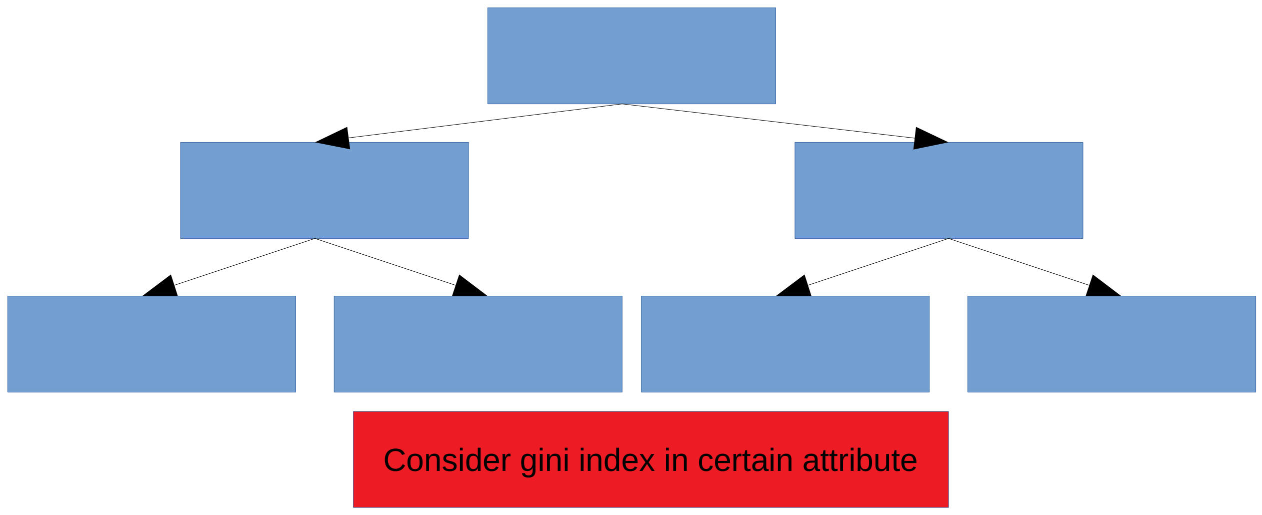
My strategy for pruning is somewhat different from conventional pruning. Conventional pruning method limits **'depth of tree'** by removing branch. However, my pruning strategy regulates **'maximum number of partitions'** in each tree depth.

I use Information gain for attribute selection and gini index for partition. Conventional gini index causes over fitting problem as it considers every binary partitions. However I could overcome this problem by using information gain for attribute selection. In specified attribute which is selected by using information gain, there are no such difficulty to dealing with large number of choices.

Suppose data set have k attributes and each attribute has atmost m number of values. In conventional gini index method, it has to consider $_{(m*k)}C_{(2)}$ conditions. In contrast, my strategy only considers $_{m}C_{2}$ conditions. In this characteristics, I think this strategy is reasonable.



[Figure1: Conventional pruning]



[Figure2: my pruning strategy]

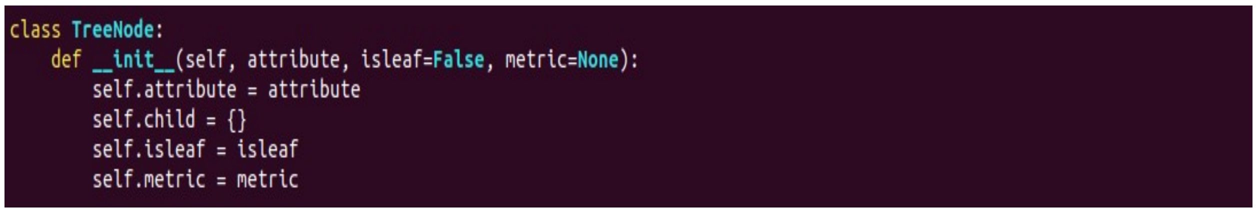
5. Analysis of code

i) Data structures

I use **'pandas data frame'** to analyze data set effectively. By using this, I can use query statement conveniently.

I use two class for representing **'Decision tree'** and **'Decision tree node'**.

a. Tree node class

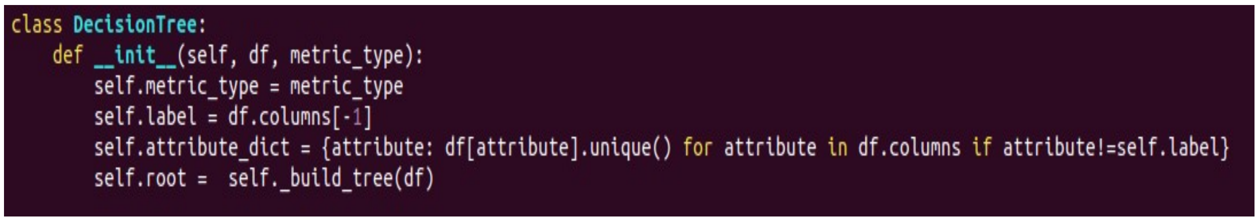


```
class TreeNode:
    def __init__(self, attribute, isleaf=False, metric=None):
        self.attribute = attribute
        self.child = {}
        self.isleaf = isleaf
        self.metric = metric
```

Tree node class has four class attributes.

1. attribute: attribute saves selected attribute of certain node.
2. child: child saves children nodes of certain node.
3. isleaf: isleaf represents whether it's leaf node or not.
4. metric: metric saves class label of certain leaf node. Only leaf node has metric attribute, otherwise none.

b. Decision tree class



```
class DecisionTree:
    def __init__(self, df, metric_type):
        self.metric_type = metric_type
        self.label = df.columns[-1]
        self.attribute_dict = {attribute: df[attribute].unique() for attribute in df.columns if attribute!=self.label}
        self.root =  self._build_tree(df)
```

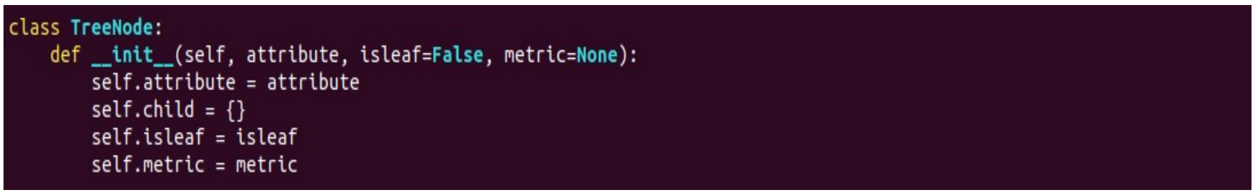
Decision tree class has four class attributes.

1. metric_type: metric_type determine attribute selection strategy. '0' denotes information gain, '1' denotes gain ratio.
2. label: label saves class label column name.
3. attribute_dict: attribute_dict saves attributes and values of each attribute of data set.
4. root: root saves root node of decision tree

ii) Functions

I use 1 function in **'Tree node'** class.

- __init__



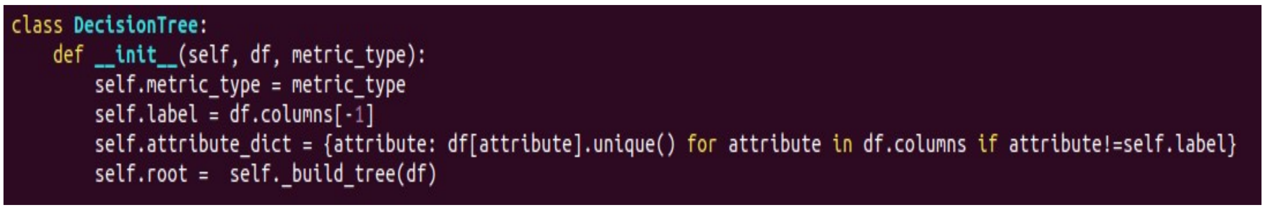
```
class TreeNode:
    def __init__(self, attribute, isleaf=False, metric=None):
        self.attribute = attribute
        self.child = {}
        self.isleaf = isleaf
        self.metric = metric
```

Feature: Create new tree node
Arguments: attribute → string, isleaf → boolean, metric → string
Return value: None

In this function, tree node is created.

I use 14 functions in 'Decision tree' class.

-__init__

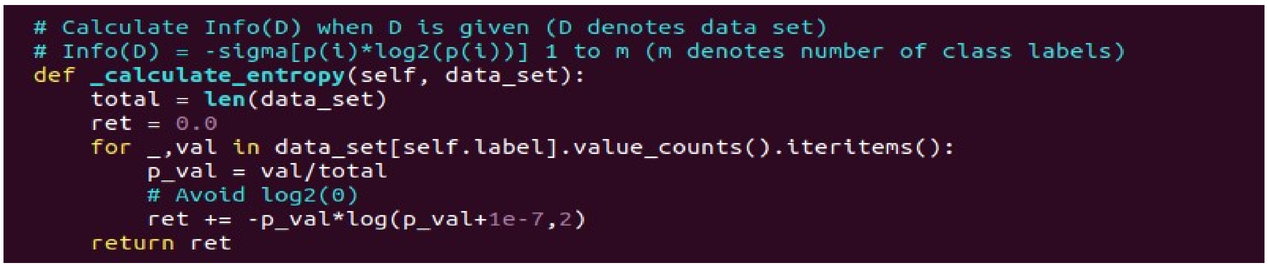


```
class DecisionTree:
    def __init__(self, df, metric_type):
        self.metric_type = metric_type
        self.label = df.columns[-1]
        self.attribute_dict = {attribute: df[attribute].unique() for attribute in df.columns if attribute!=self.label}
        self.root =  self._build_tree(df)
```

Feature: Create new decision tree
Arguments: df → pandas data frame, metric_type → integer
Return value: None

In this function, decision tree is created. Its root node is also created by using '_build_tree' function.

-_calculate_information_gain

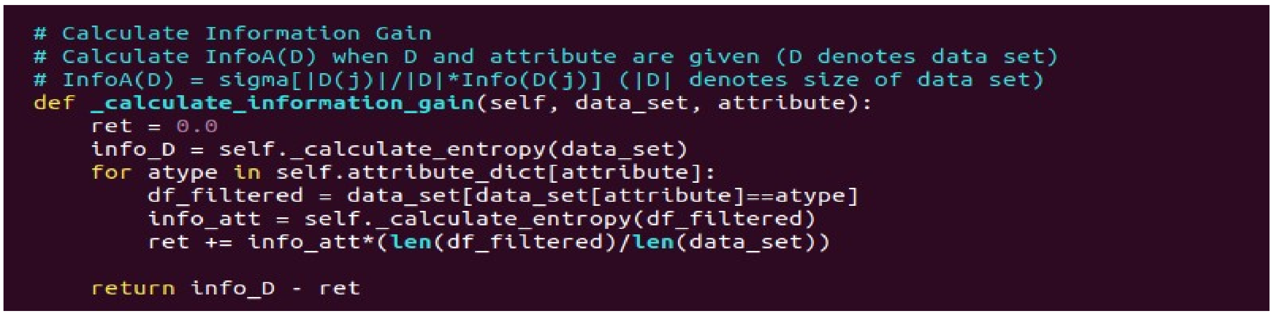


```
    # Calculate Info(D) when D is given (D denotes data set)
    # Info(D) = -sigma[p(i)*log2(p(i))] 1 to m (m denotes number of class labels)
    def _calculate_entropy(self, data_set):
        total = len(data_set)
        ret = 0.0
        for _,val in data_set[self.label].value_counts().iteritems():
            p_val = val/total
            # Avoid log2(0)
            ret += -p_val*log(p_val+1e-7,2)
        return ret
```

Feature: Calculate entropy of given data set
Arguments: data_set→ pandas data frame
Return value: ret → float

In this function, entropy of given data set is calculated. Using pandas 'value_counts' function, we can extract number of each class label counts. In usage of log, 1e-7 is added for avoiding zero input in logarithm.

-_calculate_entropy



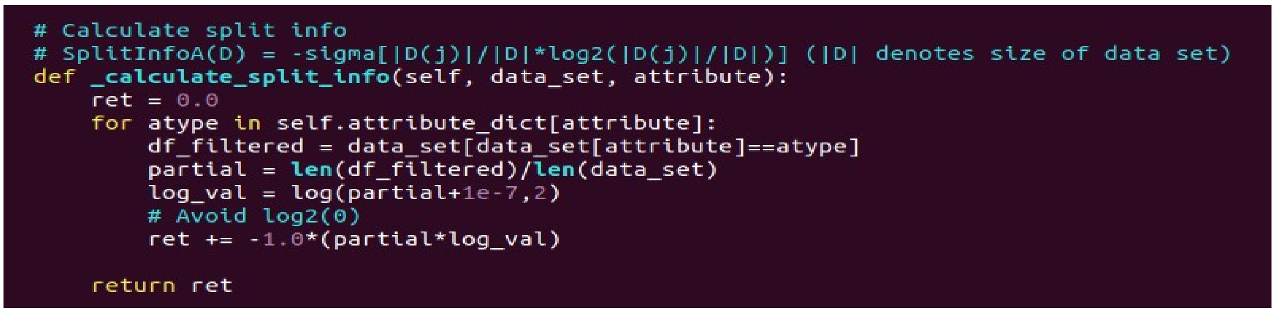
```python
# Calculate Information Gain
# Calculate InfoA(D) when D and attribute are given (D denotes data set)
# InfoA(D) = sigma[|D(j)|/|D|*Info(D(j)] (|D| denotes size of data set)
def _calculate_information_gain(self, data_set, attribute):
    ret = 0.0
    info_D = self._calculate_entropy(data_set)
    for atype in self.attribute_dict[attribute]:
        df_filtered = data_set[data_set[attribute]==atype]
        info_att = self._calculate_entropy(df_filtered)
        ret += info_att*(len(df_filtered)/len(data_set))

    return info_D - ret
```

Feature: Calculate information gain of given data set
Arguments: data_set→ pandas data frame, attribute → string
Return value: info_D - ret → float

In this function, information gain of given data set is calculated.
First, calculate entropy of original data set.
Second, calculate entropy of subset in weighted average method.
Then subtract first from second to produce information gain.

-_calculate_split_info



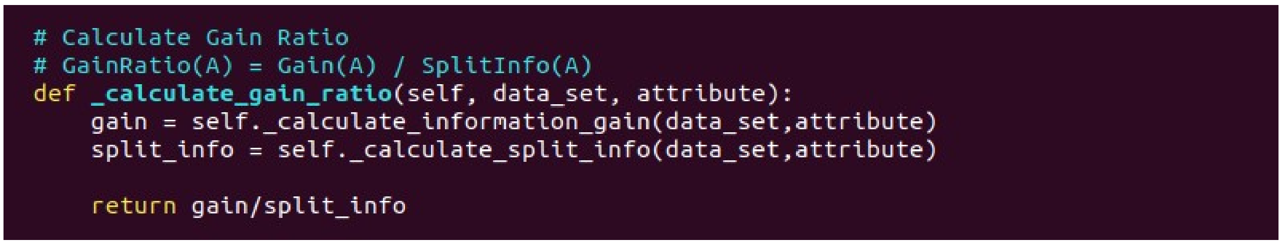
```python
# Calculate split info
# SplitInfoA(D) = -sigma[|D(j)|/|D|*log2(|D(j)|/|D|)] (|D| denotes size of data set)
def _calculate_split_info(self, data_set, attribute):
    ret = 0.0
    for atype in self.attribute_dict[attribute]:
        df_filtered = data_set[data_set[attribute]==atype]
        partial = len(df_filtered)/len(data_set)
        log_val = log(partial+1e-7,2)
        # Avoid log2(0)
        ret += -1.0*(partial*log_val)

    return ret
```

Feature: Calculate split information of given data set and attribute
Arguments: data_set→ pandas data frame, attribute → string
Return value: ret → float

In this function, split information of given data set and attribute is calculated. Pandas query statement is used for effective data access. In usage of log, 1e-7 is added for avoiding zero input in logarithm.

-_calculate_gain_ratio



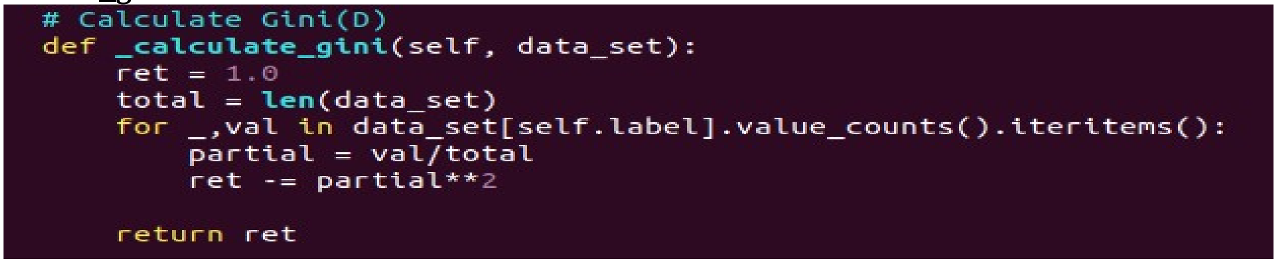
```python
# Calculate Gain Ratio
# GainRatio(A) = Gain(A) / SplitInfo(A)
def _calculate_gain_ratio(self, data_set, attribute):
    gain = self._calculate_information_gain(data_set,attribute)
    split_info = self._calculate_split_info(data_set,attribute)

    return gain/split_info
```

Feature: Calculate gain ratio of given data set and attribute
Arguments: data_set→ pandas data frame, attribute → string
Return value: gain/split_info → float

In this function, gain ration of given data set and attribute is calculated.
'_calculate_information_gain' and **'_calculate_split_info'** functions are used.

-_calculate_gini



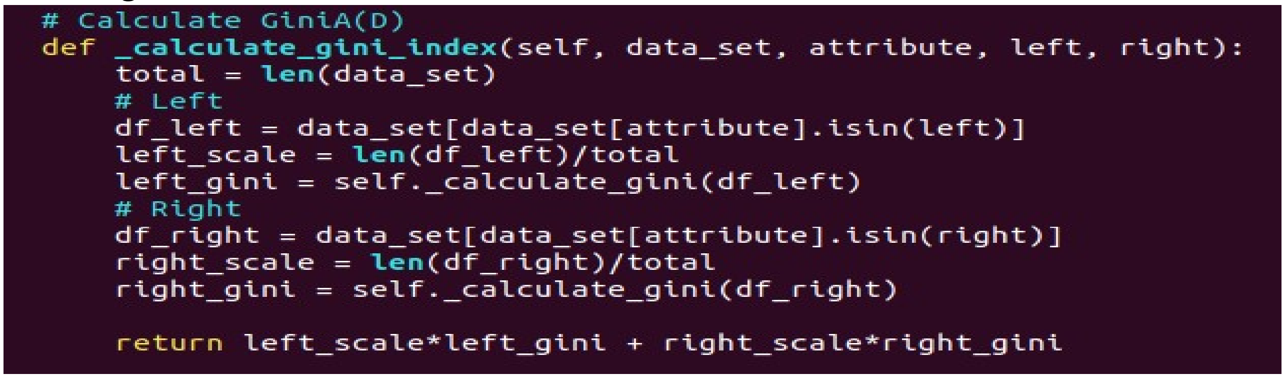
```python
# Calculate Gini(D)
def _calculate_gini(self, data_set):
    ret = 1.0
    total = len(data_set)
    for _,val in data_set[self.label].value_counts().iteritems():
        partial = val/total
        ret -= partial**2

    return ret
```

Feature: Calculate gini value for given data set
Arguments: data_set→ pandas data frame
Return value: ret → float

In this function, gini value is calculated. By using pandas ‘value_counts’, ‘iteritems’, we can access data conveniently.

-_calculate_gini_index



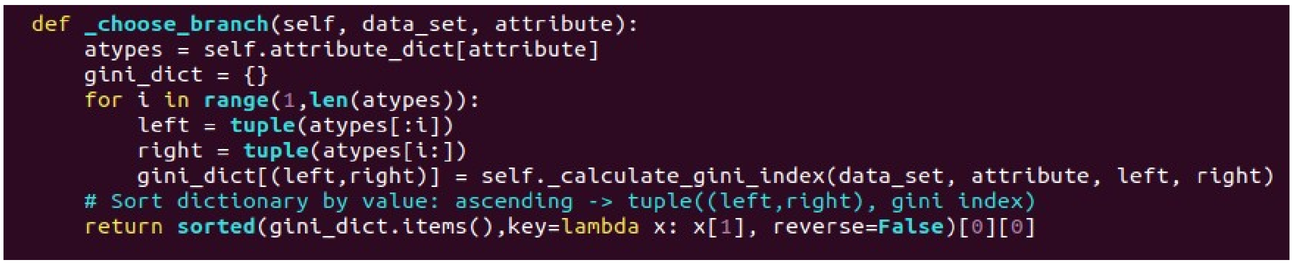
```python
# Calculate GiniA(D)
def _calculate_gini_index(self, data_set, attribute, left, right):
    total = len(data_set)
    # Left
    df_left = data_set[data_set[attribute].isin(left)]
    left_scale = len(df_left)/total
    left_gini = self._calculate_gini(df_left)
    # Right
    df_right = data_set[data_set[attribute].isin(right)]
    right_scale = len(df_right)/total
    right_gini = self._calculate_gini(df_right)

    return left_scale*left_gini + right_scale*right_gini
```

Feature: Calculate gini index for given data set and attribute
Arguments: data_set→ pandas data frame, attribute → string, left → list, right → list
Return value: left_scale*left_gini + right_scale*right_gini→ float

In this function, gini index for given data set and attribute is calculated. As gini index supposes binary partition, we have two subsets(left and right). Each subset’s gini value is calculated by using ‘_calculate_gini’ function and multiplied with its scale factor. Then sum of them is returned.

-_choose_branch

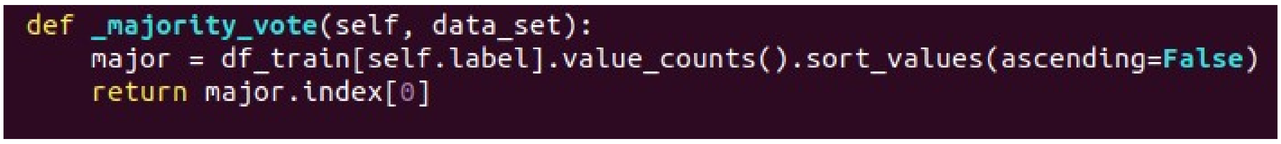


```python
def _choose_branch(self, data_set, attribute):
    atypes = self.attribute_dict[attribute]
    gini_dict = {}
    for i in range(1,len(atypes)):
        left = tuple(atypes[:i])
        right = tuple(atypes[i:])
        gini_dict[(left,right)] = self._calculate_gini_index(data_set, attribute, left, right)
    # Sort dictionary by value: ascending -> tuple((left,right), gini index)
    return sorted(gini_dict.items(),key=lambda x: x[1], reverse=False)[0][0]
```

Feature: Choose proper branch for given data set and attribute
Arguments: data_set→ pandas data frame, attribute → string
Return value: sorted(gini_dict.items(),key=lambda x:x[1], reverse=False)[0][0] → tuple

In this function, proper branch for given data set and attribute is chosen. In certain attribute, we can make every two subsets by using the attribute’s values. Then calculate gini index of each candidate(two subsets). Finally choose smallest one, which denotes largest reduction in impurity, and return it.

-_majority_vote

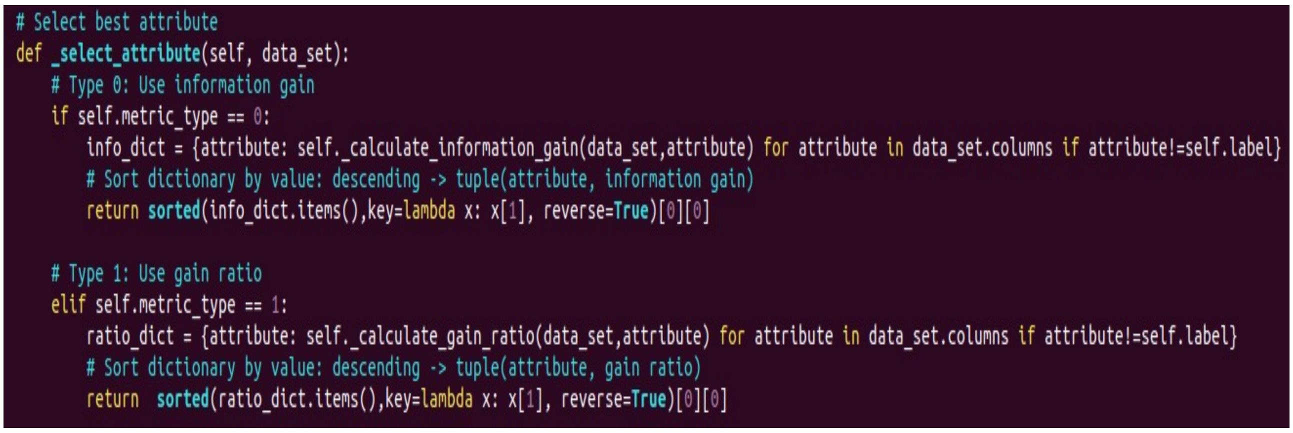


```
def _majority_vote(self, data_set):
    major = df_train[self.label].value_counts().sort_values(ascending=False)
    return major.index[0]
```

Feature: Return proper class label for given data set
Arguments: data_set→ pandas data frame
Return value: major.index[0] → string

In this function, proper class label for given data set is returned. By sorting each class label's number of counts, we can know the most frequently one. Then return it.

-_select_attribute



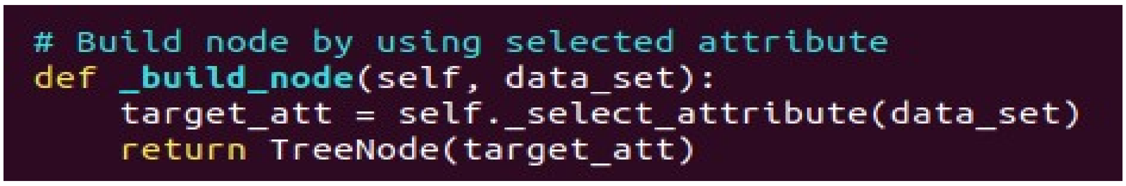
```
# Select best attribute
def _select_attribute(self, data_set):
    # Type 0: Use information gain
    if self.metric_type == 0:
        info_dict = {attribute: self._calculate_information_gain(data_set,attribute) for attribute in data_set.columns if attribute!=self.label}
        # Sort dictionary by value: descending -> tuple(attribute, information gain)
        return sorted(info_dict.items(),key=lambda x: x[1], reverse=True)[0][0]

    # Type 1: Use gain ratio
    elif self.metric_type == 1:
        ratio_dict = {attribute: self._calculate_gain_ratio(data_set,attribute) for attribute in data_set.columns if attribute!=self.label}
        # Sort dictionary by value: descending -> tuple(attribute, gain ratio)
        return  sorted(ratio_dict.items(),key=lambda x: x[1], reverse=True)[0][0]
```

Feature: Return proper attribute for given data set
Arguments: data_set→ pandas data frame
Return value: sorted(info_dict.items(),key=lambda x:x[1],reverse=True)[0][0]→ string
sorted(ratio_dict.items(),key=lambda x:x[1],reverse=True)[0][0]→ string

In this function, proper attribute for given data set is returned. Using **'metric_type'** value, we can choose attribute selection method. **'Type 0'** denotes information gain method and **'Type 1'** denotes gain ratio.

-_build_node

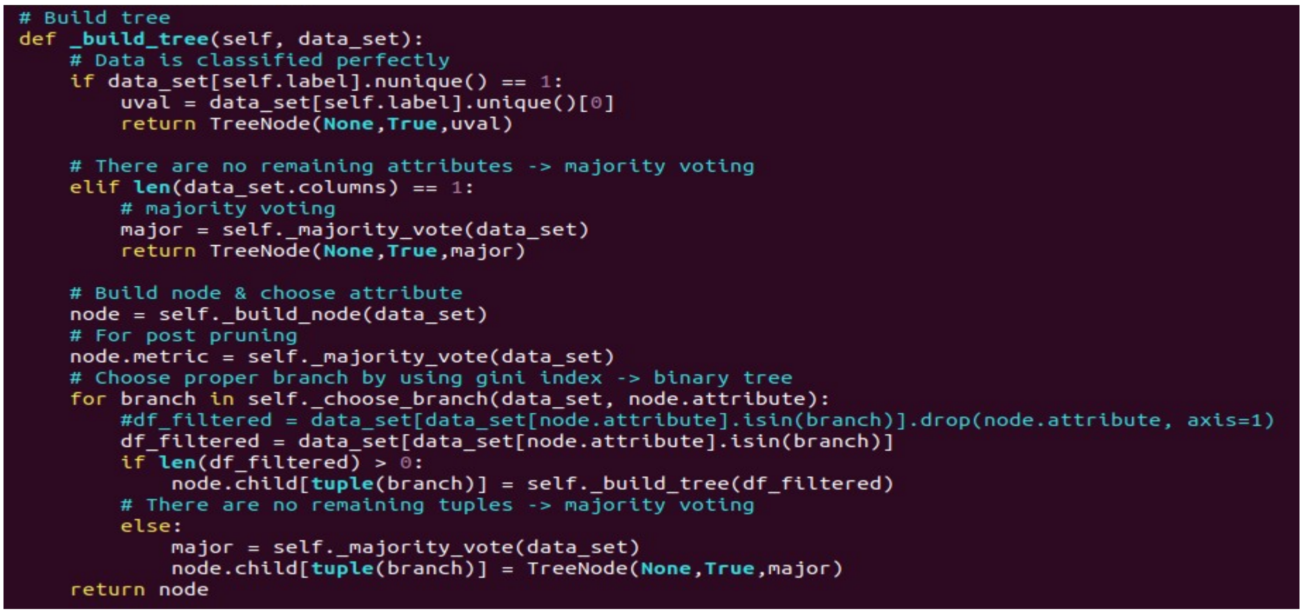


```
# Build node by using selected attribute
def _build_node(self, data_set):
    target_att = self._select_attribute(data_set)
    return TreeNode(target_att)
```

Feature: Create node for given data set
Arguments: data_set→ pandas data frame
Return value: TreeNode(target_att) → object(class TreeNode)

In this function, node is created for given data. Using **'_select_attribute'** function, proper attribute is selected. Then new node is created with proper attribute.

-_build_tree



```
# Build tree
def _build_tree(self, data_set):
    # Data is classified perfectly
    if data_set[self.label].nunique() == 1:
        uval = data_set[self.label].unique()[0]
        return TreeNode(None,True,uval)

    # There are no remaining attributes -> majority voting
    elif len(data_set.columns) == 1:
        # majority voting
        major = self._majority_vote(data_set)
        return TreeNode(None,True,major)

    # Build node & choose attribute
    node = self._build_node(data_set)
    # For post pruning
    node.metric = self._majority_vote(data_set)
    # Choose proper branch by using gini index -> binary tree
    for branch in self._choose_branch(data_set, node.attribute):
        #df_filtered = data_set[data_set[node.attribute].isin(branch)].drop(node.attribute, axis=1)
        df_filtered = data_set[data_set[node.attribute].isin(branch)]
        if len(df_filtered) > 0:
            node.child[tuple(branch)] = self._build_tree(df_filtered)
        # There are no remaining tuples -> majority voting
        else:
            major = self._majority_vote(data_set)
            node.child[tuple(branch)] = TreeNode(None,True,major)
    return node
```

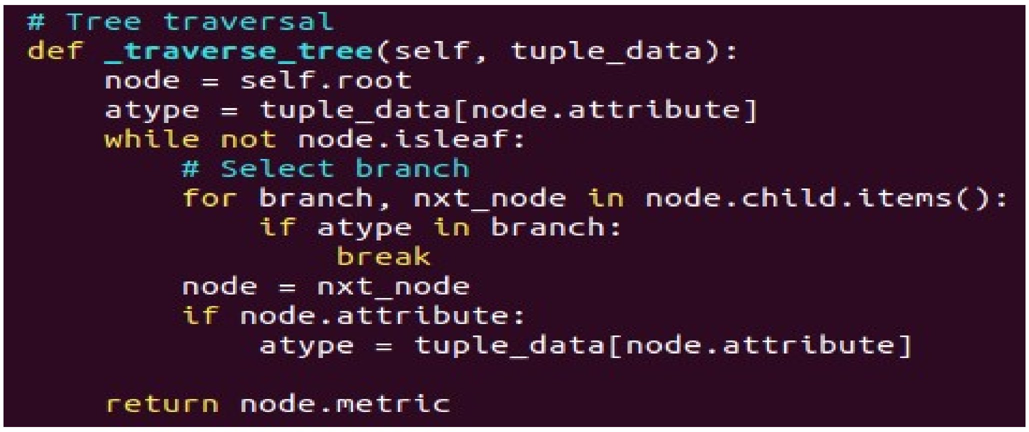
Feature: Build tree for given data set
Arguments: data_set→ pandas data frame
Return value: node → object(class TreeNode)

In this function, tree is built for given data set. This function works 'recursively'.
This is the end points of recursion

1. When given data set has unique class label, which means data set is classified perfectly, return leaf node.
2. When given data set has no remaining attributes, majority voting is done and return leaf node. For higher accuracy, this statement is not used because I don't erase any attribute columns.
   Note: I commented follow statement: df_filtered=data_set[data_set[node.attribute].isin(branch)].drop(node.attribute, axis=1)
3. When there are no remaining tuple in given data set, majority voting is done and return leaf node.

Otherwise, split data for chosen left and right branch and build tree for filtered data set recursively.

-_traverse_tree



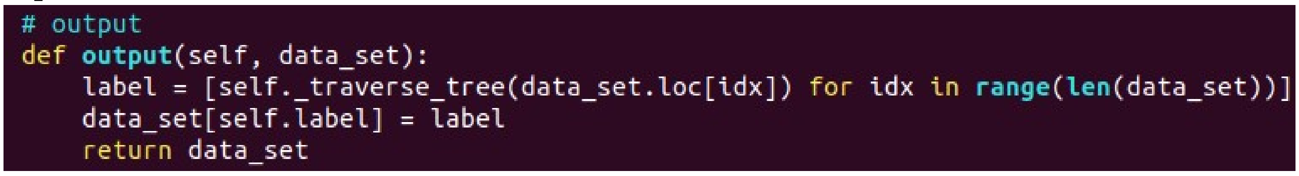
```
# Tree traversal
def _traverse_tree(self, tuple_data):
    node = self.root
    atype = tuple_data[node.attribute]
    while not node.isleaf:
        # Select branch
        for branch, nxt_node in node.child.items():
            if atype in branch:
                break
        node = nxt_node
        if node.attribute:
            atype = tuple_data[node.attribute]

    return node.metric
```

Feature: Traverse tree until it reaches leaf node for given tuple data
Arguments: tuple_data→ pandas data frame
Return value: node.metric → string

In this function, tuple data traverses tree until it reaches leaf node. Leaf node's metric is returned to classify tuple data.

-output

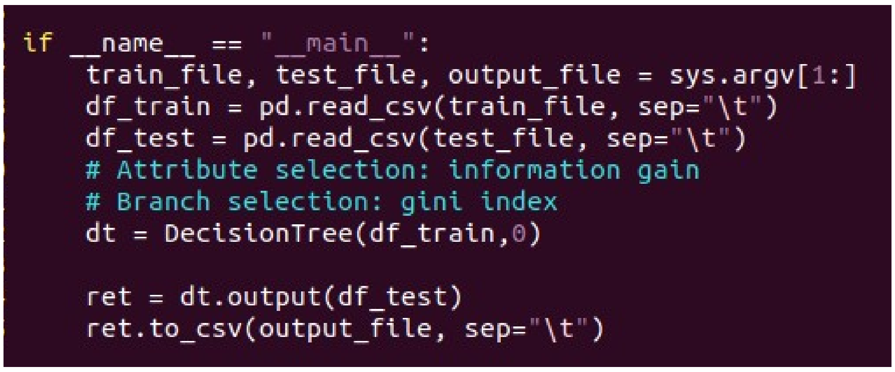


```
# output
def output(self, data_set):
    label = [self._traverse_tree(data_set.loc[idx]) for idx in range(len(data_set))]
    data_set[self.label] = label
    return data_set
```

Feature: Classify given data set
Arguments: data_set→ pandas data frame
Return value: data_set → pandas data frame

In this function, data set is classified. By using **'_traverse_tree'** function in each tuple data, we can get class label which is predicted by decision tree. Then this label is added to given data and return it.

iii) Code flow



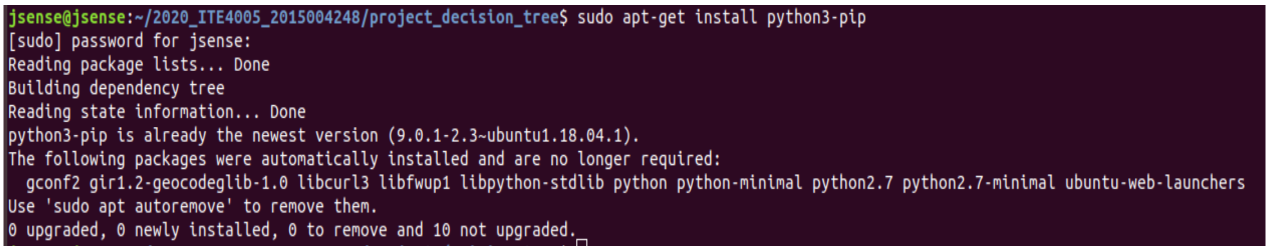
```
if __name__ == "__main__":
    train_file, test_file, output_file = sys.argv[1:]
    df_train = pd.read_csv(train_file, sep="\t")
    df_test = pd.read_csv(test_file, sep="\t")
    # Attribute selection: information gain
    # Branch selection: gini index
    dt = DecisionTree(df_train,0)

    ret = dt.output(df_test)
    ret.to_csv(output_file, sep="\t")
```

1. Save command-line arguments.
2. By using pandas **'read_csv'** function, transform train data and test data into pandas data frame.
3. Construct decision tree.
4. By using constructed decision tree, classify test data.
5. Write the result by using pandas **'to_csv'** function

6. Instructions for execution

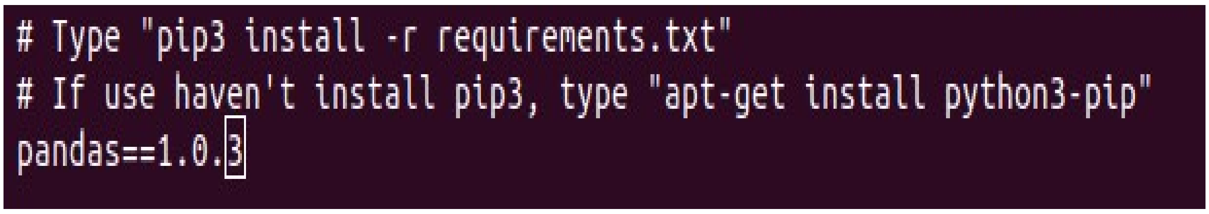
As I'm using external library(pandas), we need to install it.
First, check for **'pip3'**. If you haven't install pip3, install if first. It requires **'sudo permission'**.



```
jsense@jsense:~/2020_ITE4005_2015004248/project_decision_tree$ sudo apt-get install python3-pip
[sudo] password for jsense:
Reading package lists... Done
Building dependency tree
Reading state information... Done
python3-pip is already the newest version (9.0.1-2.3~ubuntu1.18.04.1).
The following packages were automatically installed and are no longer required:
  gconf2 gir1.2-geocodeglib-1.0 libcurl3 libfwup1 libpython-stdlib python python-minimal python2.7 python2.7-minimal ubuntu-web-launchers
Use 'sudo apt autoremove' to remove them.
0 upgraded, 0 newly installed, 0 to remove and 10 not upgraded.
```

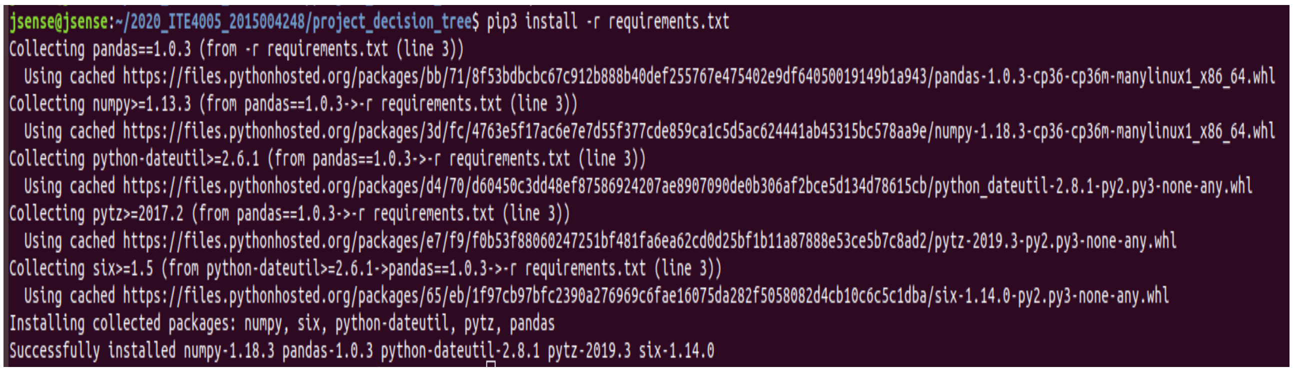
Type 'sudo apt-get install python3-pip' in terminal

Then install python3 pandas library using pip3. I made **‘requirements.txt’** for convenient installation. It’s located in same directory with **‘dt.py’**



```
# Type "pip3 install -r requirements.txt"
# If use haven't install pip3, type "apt-get install python3-pip"
pandas==1.0.3
```

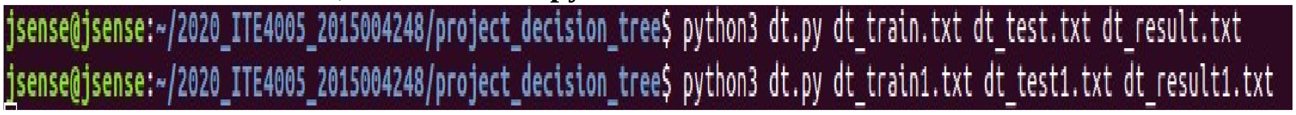
requirements.txt



```
jsense@jsense:~/2020_ITE4005_2015004248/project_decision_tree$ pip3 install -r requirements.txt
Collecting pandas==1.0.3 (from -r requirements.txt (line 3))
  Using cached https://files.pythonhosted.org/packages/bb/71/8f53bdbcbc67c912b888b40def255767e475402e9df64050019149b1a943/pandas-1.0.3-cp36-cp36m-manylinux1_x86_64.whl
Collecting numpy>=1.13.3 (from pandas==1.0.3->-r requirements.txt (line 3))
  Using cached https://files.pythonhosted.org/packages/3d/fc/4763e5f17ac6e7e7d55f377cde859ca1c5d5ac624441ab45315bc578aa9e/numpy-1.18.3-cp36-cp36m-manylinux1_x86_64.whl
Collecting python-dateutil>=2.6.1 (from pandas==1.0.3->-r requirements.txt (line 3))
  Using cached https://files.pythonhosted.org/packages/d4/70/d60450c3dd48ef87586924207ae8907090de0b306af2bce5d134d78615cb/python_dateutil-2.8.1-py2.py3-none-any.whl
Collecting pytz>=2017.2 (from pandas==1.0.3->-r requirements.txt (line 3))
  Using cached https://files.pythonhosted.org/packages/e7/f9/f0b53f88060247251bf481fa6ea62cd0d25bf1b11a87888e53ce5b7c8ad2/pytz-2019.3-py2.py3-none-any.whl
Collecting six>=1.5 (from python-dateutil>=2.6.1->pandas==1.0.3->-r requirements.txt (line 3))
  Using cached https://files.pythonhosted.org/packages/65/eb/1f97cb97bfc2390a276969c6fae16075da282f5058082d4cb10c6c5c1dba/six-1.14.0-py2.py3-none-any.whl
Installing collected packages: numpy, six, python-dateutil, pytz, pandas
Successfully installed numpy-1.18.3 pandas-1.0.3 python-dateutil-2.8.1 pytz-2019.3 six-1.14.0
```

Type ‘pip3 install -r requirements.txt’ in terminal

After successful installation, we can run **‘dt.py’**

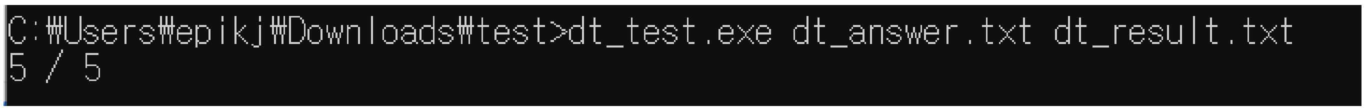


```
jsense@jsense:~/2020_ITE4005_2015004248/project_decision_tree$ python3 dt.py dt_train.txt dt_test.txt dt_result.txt
jsense@jsense:~/2020_ITE4005_2015004248/project_decision_tree$ python3 dt.py dt_train1.txt dt_test1.txt dt_result1.txt
```

Type ‘dt.py “training file name” “test file name” “output file name”’ in terminal

7. Test result

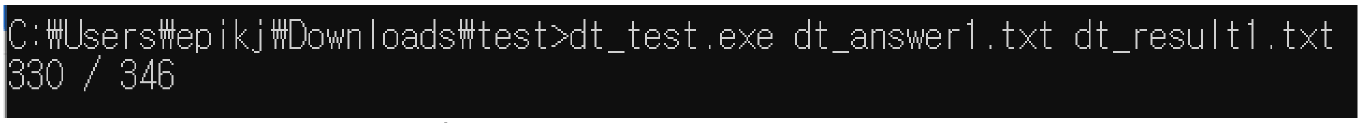
i) Result of 1st data set



```
C:₩Users₩epikj₩Downloads₩test>dt_test.exe dt_answer.txt dt_result.txt
5 / 5
```

I got 100% accuracy from 1st data set.

ii) Result of 2nd data set



```
C:₩Users₩epikj₩Downloads₩test>dt_test.exe dt_answer1.txt dt_result1.txt
330 / 346
```

I got about 95% accuracy from 2nd data set.

8. Reference

[1] Jiawei Han. Data mining: concepts and techniques(3rd edition), 2000.