Intelligent Data Analysis – Spring 2018
Homework #3

- Consider the two data files mentioned below for this homework assignment. These data files are from the UCI Machine Learning Repository. The first file is for the Breast Cancer Prognostics (BCP) data set and is available at the link: https://archive.ics.uci.edu/ml/datasets/Breast+Cancer+Wisconsin+(Prognostic) . The second dataset is about wine quality (WQ) and is available at: https://archive.ics.uci.edu/ml/datasets/wine+quality . It has two separate datasets, one for red wine quality (RWQ) and the other for white wine quality (WWQ) . Perform the following tasks with these datasets and submit the answers asked for in each task listed below.
- All your answers must be contained in a single pdf file. Upload this pdf file on Blackboard in response to the homework assignment.
- Each submission must be individual work of a student. Any plagiarism detected will be severely punished.

1. Consider the BCP dataset and its class variable with values "R" (Recurrence Occurred) and "N" (No Recurrence Occurred so far). Ignore the attribute that gives the number of years after which recurrence occurred or the number of years for which the patient has been free of recurrence. There are thirty other attribute values given as features measured for every patient. For these two classes of patients perform the following:

a. (14 Adjust any parameters of decision tree construction algorithm to find the decision tree with the best possible performance in terms of accuracy. Use four-fold validation for obtaining the accuracy results. You can set the four-fold validation as an option in the tool-box you use for generating the decision tree. Report the selected parameter values and the resulting optimal tree. For this decision tree find the precision and recall values of each of the two classes.

Resulted Graph:

Using 4-folded validation generated decision tree parameters:

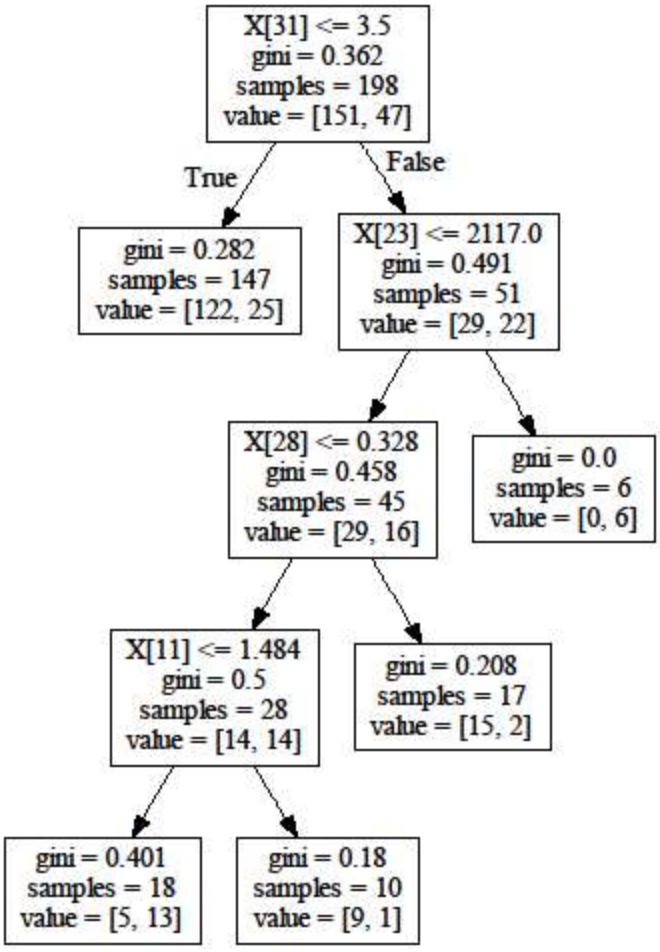
```
class_weight=None,
criterion='gini',
max_depth=None,
max_features=None,
max_leaf_nodes=5,
min_impurity_decrease=0.0,
min_impurity_split=None,
 min_samples_leaf=5,
min_samples_split=20,
min_weight_fraction_leaf=0.0,
presort=False,
random_state=None,
splitter='best'
```

```
GridSearchCV(cv=4, error_score='raise',
       estimator=DecisionTreeClassifier(class_weight=None, criterion='gini', max_depth=None,
            max_features=None, max_leaf_nodes=None,
            min_impurity_decrease=0.0, min_impurity_split=None,
            min_samples_leaf=1, min_samples_split=2,
            min_weight_fraction_leaf=0.0, presort=False, random_state=None,
            splitter='best'),
       fit_params=None, iid=True, n_jobs=1,
       param_grid={'criterion': ['gini', 'entropy'], 'min_samples_split': [2, 10, 20], 'max_depth': [None, 2, 5, 10], 'min_samp
les_leaf': [1, 5, 10], 'max_leaf_nodes': [None, 5, 10, 20]},
       pre_dispatch='2*n_jobs', refit=True, return_train_score='warn',
       scoring=None, verbose=0)
```

```
tree_model = clf.best_estimator_
```

```
tree_model
```

```
DecisionTreeClassifier(class_weight=None, criterion='gini', max_depth=None,
            max_features=None, max_leaf_nodes=5, min_impurity_decrease=0.0,
            min_impurity_split=None, min_samples_leaf=5,
            min_samples_split=20, min_weight_fraction_leaf=0.0,
            presort=False, random_state=None, splitter='best')
```




For class –N

```
precision 0.8390804597701149
Recall 0.9668874172185431
Fscore 0.8984615384615384
```

For class- R

```
precision 0.7916666666666666
Recall 0.40425531914893614
Fscore 0.5352112676056338
```

b. (6) Provide an analysis of the performance metrics in terms of the data's domain.

Over All Result:

```
print(Accuracy1, Recall1 , Precision1)
```

```
83.33333333333334 83.33333333333334 82.78256704980843
```

c. The results given nearly good performance for both of the classes with nearly 80% precision.
d. As per my analysis, this is having larger partition, may be that's why it choose gini index.
e. In the data set I found the data is pure, so the nodes can be expanded till they became pure so max depth is none.
f. For better accuracy the depth is stopped at 5

Reference:

http://scikit-learn.org/stable/modules/generated/sklearn.model_selection.KFold.html

https://stackoverflow.com/questions/35097003/cross-validation-decision-trees-in-sklearn

2. (20) Now ignore the class attribute (R or N) in BCP dataset and use the number of years (at which recurrence occurred or the years for which the patient has been free). Consider this value to be the number of years for which a patient is expected to remain disease-free. We want to build a regression tree to predict this number - the number of disease-free years for each patient. Do not use any in-built regression function for building the regression tree. Use your own code for each individual step towards building the regression tree. These steps can be executed by you in any programming environment or tool box. Show the code and results for each step. The steps to be performed are:

a. Assume the average number of disease-free years for the entire training dataset as the initial predicted value for each record of the dataset. Find the mean square error (MSE) of making this prediction for all the training and the test set records (separately for training and test sets).

```
#read csv files of 2 data sets
my_data = pd.read_csv("C:/Users/mereddda/Desktop/IDA/breast-cancer-wisconsin.csv", skipinitialspace=True)
my_data.drop(my_data.columns[[1]], axis=1, inplace=True)
my_data.head()
```

Split the dataset into training (2/3rd) and testing dataset (1/3rd).

```
In [13]:
X=my_data.iloc[:,2:]
Y=my_data.iloc[:,1]

X_test=X.iloc[:132,:]
X_train=X.iloc[132:,:]
Y_test=Y.iloc[:132]
Y_test
Y_train=Y.iloc[132:]
Y_train
```

```
Out[13]: 132     5
         133    61
         134    51
         135     2
         136    54
         137     7
         138    57
         139    13
```

Assume the average number of disease-free years for the entire training dataset as the initial predicted value for each record of the dataset.
Mean square error (MSE) of making this prediction for all the training and the test set records.

```
y_predict = [y_pred] * len(Y_train)
```

```
train_MSE_error = mean_squared_error(Y_train, y_predict)
```

```
train_MSE_error
```

```
298.5824150596878
```

```
y_predict = [y_pred] * len(Y_test)
test_MSE_error  = mean_squared_error(Y_test, y_predict)
test_MSE_error
```

```
2501.851928374656
```

a. Find the attribute most closely correlated to the number of disease-free years in the training data; take the median value of this attribute as the split-point and partition the dataset into two parts.

```
X_train_sample = X_train;
```

```
correlation = X_train_sample.apply(lambda x: x.corr(Y_train))
max_col = correlation.idxmax()
col_med = st.median(X_train.loc[:,max_col])
```

```
col_med
```

```
0.28205
```

```
datax = np.where(X_train.loc[:,max_col] > col_med)
```

```
datax
```

```
(array([ 2,  5,  8,  9, 10, 11, 13, 14, 15, 17, 18, 22, 23, 24, 25, 28, 30,
        32, 34, 35, 36, 37, 42, 46, 48, 50, 56, 59, 60, 62, 63, 64, 65],
       dtype=int64),)
```

```
X_train_right = X_train.loc[X_train.loc[:,max_col] >= col_med]
X_train_left = X_train.loc[X_train.loc[:,max_col] < col_med]
X_train_sample = X_train_sample.drop(max_col,1)
```

```
X_train_sample
```

| | Radius_Mean | Radius_SE | Radius_Worst | Texture_Mean | Texture_SE | Texture_Worst | Perimeter_Mean | Perimeter_SE | Perimeter_Worst | Area_Mean | ... | CV_M |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 132 | 15.08 | 25.74 | 98.00 | 716.6 | 0.10240 | 0.09769 | 0.12350 | 0.06553 | 0.1647 | 0.06464 | ... | 3 |
| 133 | 20.44 | 21.78 | 133.80 | 1293.0 | 0.09150 | 0.11310 | 0.09799 | 0.07785 | 0.1618 | 0.05557 | ... | 2 |
| 134 | 20.20 | 26.83 | 133.70 | 1234.0 | 0.09905 | 0.16690 | 0.16410 | 0.12650 | 0.1875 | 0.06020 | ... | 3 |
| 135 | 22.01 | 21.90 | 147.20 | 1482.0 | 0.10630 | 0.19540 | 0.24480 | 0.15010 | 0.1824 | 0.06140 | ... | 2 |

b. Repeat step (b) recursively for each leaf node until all the leaf nodes have their MSE values below some threshold value.

In [106]:
```
def database_split(df):
    correlation =df.apply(lambda x: x.corr(Y_train))
    max_col = correlation.idxmax()
    col_med = st.median(df.loc[:,max_col])
    right = df.loc[df.loc[:,max_col] >= col_med]
    left = df.loc[df.loc[:,max_col] < col_med]
    df = df.drop(max_col,1)
    return left, right #(left df, right df
database_split(X_train)
```

Out[106]:
```
(     Radius_Mean  Radius_SE  Radius_Worst  Texture_Mean  Texture_SE  \
 132        15.08      25.74         98.00         716.6     0.10240
 133        20.44      21.78        133.80        1293.0     0.09150
 135        22.01      21.90        147.20        1482.0     0.10630
```

In [109]:
```
def MSE_error(df):
    y_pred = df.mean()
    y_predict = [y_pred] * len(df)
    train_MSE_error = mean_squared_error(df, y_predict)
    return train_MSE_error
MSE_error(Y_train)
```

Out[109]:
```
298.5824150596878
```

Decide on a suitable threshold to stop growing the regression tree. Explain how you chose this threshold.

1. Initially I started with a near value of mean MSB I got I the above step that is 298.

2. Then repeatedly run the code and changed the thresholds to get better results with less MSB.

3. When I found the length of the decision tree is increasing more I stopped decreasing threshold value though the MSB slightly decreases with that.

In [463]:
```
#read CSV files of 2 data sets
my_data = pd.read_csv("C:/Users/mereddda/Desktop/IDA/breast-cancer-wisconsin.csv", skipinitialspace=True)
my_data.drop(my_data.columns[[1]], axis=1, inplace=True)
my_data.drop(my_data.columns[[0]], axis=1, inplace=True)
my_data.head()
```

Out[463]:

| | Time | Radius_Mean | Radius_SE | Radius_Worst | Texture_Mean | Texture_SE | Texture_Worst | Perimeter_Mean | Perimeter_SE | Perimeter_Worst | ... | CV_SE | CV_\ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 31 | 18.02 | 27.60 | 117.50 | 1013.0 | 0.09489 | 0.1036 | 0.1086 | 0.07055 | 0.1865 | ... | 139.70 | 1 |
| 1 | 61 | 17.99 | 10.38 | 122.80 | 1001.0 | 0.11840 | 0.2776 | 0.3001 | 0.14710 | 0.2419 | ... | 184.60 | 2 |
| 2 | 116 | 21.37 | 17.44 | 137.50 | 1373.0 | 0.08836 | 0.1189 | 0.1255 | 0.08180 | 0.2333 | ... | 159.10 | 1 |
| 3 | 123 | 11.42 | 20.38 | 77.58 | 386.1 | 0.14250 | 0.2839 | 0.2414 | 0.10520 | 0.2597 | ... | 98.87 | |
| 4 | 27 | 20.29 | 14.34 | 135.10 | 1297.0 | 0.10030 | 0.1328 | 0.1980 | 0.10430 | 0.1809 | ... | 152.20 | 1 |

5 rows × 33 columns

In [478]:
```
my_data.head()
X=my_data.iloc[:,1:]
Y=my_data.iloc[:,0]

X_test=X.iloc[:132,:]
X_train=X.iloc[132:,:]
Y_test=Y.iloc[:132]
Y_test
Y_train=Y.iloc[132:]
#Y_train
#X_train
```

In [479]:
```
# Split a dataset based on an attribute and an attribute value
def test_split(index, value, dataset):
    left, right = list(), list()
    for row in dataset:
        if row[index] < value:
            left.append(row)
        else:
            right.append(row)
    return left, right
```

In [480]:
```
def df_split(df, feature, value):
 left = df[df[feature]<=value]
 right = df[df[feature]>value]
 return left, right #
```

In [481]:
```
# Select the best split point for a dataset
def get_split(dataset):
    X=dataset.iloc[:,1:]
    Y=dataset.iloc[:,0]
    b_index, b_value, b_score, b_groups = 999, 999, 999, None
    correlation =X.apply(lambda x: x.corr(Y))
    max_col = correlation.idxmax()

    col_med = st.median(dataset.loc[:,max_col])
    right = dataset.loc[dataset.loc[:,max_col] >= col_med]
    left = dataset.loc[dataset.loc[:,max_col] < col_med]
#     groups=df_split(dataset, max_col, col_med)
#     print(type(groups))
#     tup=(4,5)
#     print(type(tup))
    b_groups=(right,left)
#     dataset = dataset.drop(max_col,1)
#      return left, right #(left df, right df
    return {'index':max_col, 'value':col_med, 'groups':b_groups}
```

In [482]:
```
# Create a terminal node value
def to_terminal(group):
    outcomes = group.loc[:,'Time'].mean()
    return outcomes
```

In [483]:
```
# Create child splits for a node or make terminal
def split(node):
 left, right = node['groups']
 del(node['groups'])
 # check for a no split
 if len(left) == 0 or len(right) == 0 :
   return
 # process left child
 left_MSE = MSE_error(left)
 right_error = MSE_error(right)
 if left_MSE <= 1500:
   node['left'] = to_terminal(left)
 else:
   node['left'] = get_split(left)
   split(node['left'])
 # process right child
 if right_error <= 1500:
   node['right'] = to_terminal(right)
 else:
   node['right'] = get_split(right)
   split(node['right'])
```

In [500]:
```
def build_tree(train):
    root = get_split(train) # best split
    split(root) # split add to func
    return root
```

In [518]:
```
# Print a decision tree
def print_tree(node, depth=0):
    if isinstance(node, dict):
        print('%s[X%d < %.3f]' % ((depth*' ', (node['index']+1), node['value'])))
        print_tree(node['left'], depth+1)
        print_tree(node['right'], depth+1)
    else:
        print('%s[%s]' % ((depth*' ', node)))
```

In [520]:
```
#my_data.drop(my_data.columns[[0]], axis=1, inplace=True)
my_data.head()
tree = build_tree(my_data)
print(tree)
```

```
{'index': 'FD_Worst', 'value': 0.08689, 'left': {'index': 'FD_SE', 'value': 0.3415, 'left': {'index': 'FD_SE', 'value': 0.3920
5, 'left': {'index': 'Perimeter_Worst', 'value': 0.231, 'left': {'index': 'Symmetry_Mean', 'value': 0.1534, 'left': {'index':
'Symmetry_Mean', 'value': 0.1592, 'left': {'index': 'Texture_SE', 'value': 0.12285, 'left': 121.0, 'right': 63.5}, 'right': {'i
ndex': 'Tumor_Size', 'value': 3.0, 'left': 106.5, 'right': 58.0}}, 'right': {'index': 'Texture_Mean', 'value': 904.2, 'left': 5
9.333333333333336, 'right': 49.666666666666664}}, 'right': {'index': 'FD_SE', 'value': 0.4267, 'left': {'index': 'Tumor_Size',
'value': 2.75, 'left': 88.33333333333333, 'right': 73.0}, 'right': {'index': 'FD_SE', 'value': 0.41215, 'left': {'index': 'Peri
meter_SE', 'value': 0.07364, 'left': 69.5, 'right': 7.0}, 'right': 36.333333333333336}}}, 'right': {'index': 'FD_SE', 'value':
0.3651, 'left': {'index': 'FD_SE', 'value': 0.3706, 'left': {'index': 'FD_SE', 'value': 0.3792, 'left': {'index': 'Area_Worst',
'value': 0.9073, 'left': {'index': 'Area_Worst', 'value': 0.9529000000000001, 'left': 125.0, 'right': 58.0}, 'right': {'index':
'Texture_SE', 'value': 0.11285, 'left': 111.0, 'right': 8.0}}, 'right': {'index': 'Tumor_Size', 'value': 2.6, 'left': {'index':
'Concavity_Mean', 'value': 0.01704, 'left': 114.0, 'right': 10.0}, 'right': 3.0}}, 'right': {'index': 'Lymph_Node_Status', 'val
ue': 0.0}}, 'right': {'index': 'Smoothness_Worst', 'value': 0.005919000000000001, 'left': {'index': 'Tumor_Size', 'value': 2.0,
```

In [515]:
```
# Make a prediction with a decision tree

def predict(node, row):
#     print(node['right'])
    if row[node['index']] < node['value']:
        if isinstance(node['left'], dict):
            return predict(node['left'], row)
        else:
            return node['left']

    else:
        if isinstance(node['right'], dict):
            return predict(node['right'], row)
        else:
            return node['right']
```

In [516]:
```
test=X_test
for i in range(len(test)):
    prediction = predict(tree, test.iloc[i])
    print(prediction);
```

```
121.0
62.0
62.0
12.0
121.0
62.0
28.0
12.0
12.0
12.0
121.0
12.0
121.0
12.0
121.0
12.0
62.0
```

Tree:

```
#print(tree)
print_tree(tree)
```

```
[XFD_Worst < 0.087]
 [XFD_SE < 0.342]
  [XFD_SE < 0.392]
   [XPerimeter_Worst < 0.231]
    [XSymmetry_Mean < 0.153]
     [XSymmetry_Mean < 0.159]
      [XTexture_SE < 0.123]
       [XTexture_SE < 0.135]
        [123.0]
        [119.0]
       [XRadius_SE < 15.945]
        [66.0]
        [61.0]
      [XTumor_Size < 3.000]
       [XRadius_SE < 15.945]
        [116.0]
        [97.0]
       [58.0]
     [XTexture_Mean < 904.200]
```

Full regression tree shown down at the end of the assignment.

For other test case:

```
tree
FD_Mean            -1.0
FD_SE               1.0
FD_Worst            1.0
Tumor_Size         -1.0
Lymph_Node_Status  -1.0
dtype: float64
Radius_SE

Out[954]: {'index': 'FD_Worst',
 'left': {'index': 'FD_SE',
  'left': {'index': 'FD_SE',
   'left': {'index': 'Perimeter_Worst',
    'left': {'index': 'Symmetry_Mean',
     'left': {'index': 'Symmetry_Mean',
      'left': {'index': 'Texture_SE',
       'left': 121.0,
       'right': {'index': 'Radius_SE',
        'left': 66.0,
        'right': 61.0,
        'value': 15.945},
```

In [928]: `# Print a decision tree`

Tree:

```
print_tree(tree)

[XFD_Worst < 0.087]
 [XFD_SE < 0.342]
  [XFD_SE < 0.392]
   [XPerimeter_Worst < 0.231]
    [XSymmetry_Mean < 0.153]
     [XSymmetry_Mean < 0.159]
      [XTexture_SE < 0.123]
       [121.0]
       [XRadius_SE < 15.945]
        [66.0]
        [61.0]
      [XTumor_Size < 3.000]
       [XRadius_SE < 15.945]
        [116.0]
        [97.0]
       [58.0]
    [XTexture_Mean < 904.200]
     [XTexture_Mean < 1311.000]
      [84.5]
```

Show the resulting tree and details about each leaf node including its population, predicted value, and MSE values for the training and the test datasets.

Predicted data:

```
]: test=X_test
   for i in range(len(test)):
       prediction = predict(tree, test.iloc[i])
       print(prediction);
```

```
121.0
62.0
62.0
12.0
121.0
62.0
1.0
12.0
12.0
12.0
121.0
12.0
121.0
12.0
121.0
12.0
121.0
12.0
62.0
121.0
10.0
62.0
```

Code:

```
def get_split(dataset):
c=(len(left),len(right))
   Count.append(c)

def split(node):

msb=(left_MSE,right_error)
 MSB_list.append(msb)
```

Population of the data at nodes:
Count():

```
(25, 25),
(12, 13),
(6, 7),
(3, 4),
(2, 2),
(1, 1),
(1, 2),
(1, 1),
```

```
In [960]: MSB_list
(360.03027318218017, 633.2630287143431),
(0.0, 0.0),
(701.5887206120168, 0.0),
(0.0, 0.0),
(1479.1129727193872, 1190.8974446623977),
(212.40896973362294, 0.0),
(479.5571305357685, 0.0),
(1586.331870859919, 2988.7731744546213),
(342.923054172713, 518.4600943002281),
(576.9626569549815, 0.0),
(0.0, 0.0),
```

For leaf node 0
Predicted value of the leaf node 121.0302
Population in the leaf node 65
MSE for this leaf

For leaf node 1
Predicted value of the leaf node 62
Population in the leaf node 15
MSE for this leaf 360.03

For leaf node 2
Predicted value of the leaf node 62
Population in the leaf node 16
MSE for this leaf 633.26

For leaf node 3
Predicted value of the leaf node 12
Population in the leaf node 4
MSE for this leaf 764.1875

For leaf node 4

Predicted value of the leaf node 121.0
Population in the leaf node 2
MSE for this leaf 701.58

For leaf node 5
Predicted value of the leaf node 62.0
Population in the leaf node 1
MSE for this leaf 0.0

For leaf node 6
Predicted value of the leaf node 1.0
Population in the leaf node 1
MSE for this leaf 0.0

For leaf node 7
Predicted value of the leaf node 12.0
Population in the leaf node 4
MSE for this leaf 0.0

For leaf node 8
Predicted value of the leaf node 12.0
Population in the leaf node 2
MSE for this leaf 1479.1129

For leaf node 9
Predicted value of the leaf node 12.0
Population in the leaf node 2
MSE for this leaf 1190.89

For leaf node 10
Predicted value of the leaf node 12.0
Population in the leaf node 16
MSE for this leaf 212.4089

c.

Provide an analysis of the results obtained by the regression tree and also of the structure of the regression tree.

- In my analysis, I found as the decision tree depth increases, the MSB decreased. We changed the threshold such that it the tree gives good results for min depth.
- The MSB for the node with more branches or having more depth one when compared with a one with less branches/depth.
- The structured increases when the MSB threshold value decrease, in turn this tree gives better results. But we took threshold such that it gives less error and doesn't over fit.

What are the best and the worst paths of this regression tree?

- As per my knowledge, the path which is shorted gives the results with less number of iterations which can be considered as best path.
- A path with more nodes/ deepest branches is considered as worth path because in this to take a decision need to go with more number of iterations.

What attributes do the predictions depend on?

The precisions mostly depend on the one with more co relations.
As calculated in the code, the division is done based on them.

Features:
FD_SE
Area_SE
Etc.

Co-relation:
correlation Radius_Mean -0.018497 Radius_SE -0.255133 Radius_Worst -0.025574 Texture_Mean -0.030286 Texture_SE -0.043535 Texture_Worst -0.045112 Perimeter_Mean -0.103626 Perimeter_SE -0.033184 Perimeter_Worst 0.006962 Area_Mean -0.007577 Area_SE -0.022353 Area_Worst -0.178467 Smoothness_Mean -0.056577 Smoothness_SE -0.061364 Smoothness_Worst 0.019814 Compactness_Mean 0.040434 Compactness_SE -0.125964 Compactness_Worst -0.069626 Concavity_Mean -0.018634 Concavity_SE -0.006137 Concavity_Worst 0.034660 CV_Mean -0.267796 CV_SE 0.002739 CV_Worst 0.012297 Symmetry_Mean 0.018456 Symmetry_SE 0.032310 Symmetry_Worst -0.058099 FD_Mean 0.038189 FD_SE 0.082753 FD_Worst 0.024257 Tumor_Size -0.114738 Lymph_Node_Status -0.112108

What are the similarities and differences between the decision tree obtained in 1(a) and this regression tree?

- When given higher threshold I found both are nearly same.
- But when given lesser threshold, the depth the regression tree increases a lot.
- In total, in my experiment, decision tree is giving optimal results with depth of 5, while the regression tree is little bit higher as shown in results.

3. (20) Consider the White Wine Quality (WWQ) dataset. The target is the quality of wine which can take a value between 0 and 10. Build the best possible regression tree that you can build by following all the steps as described in (#2) above.

   Tree-data:

```
{'index': 'pH',
 'left': {'index': 'sulphates',
  'left': {'index': 'sulphates',
   'left': {'index': 'sulphates',
    'left': 6.161016949152542,
    'right': 6.095522388059702,
    'value': 0.63},
   'right': {'index': 'pH',
    'left': 6.012383900928793,
    'right': 5.761146496815287,
    'value': 3.27},
   'value': 0.56},
  'right': {'index': 'citric acid',
   'left': {'index': 'pH',
    'left': 6.12536443148688,
    'right': 5.8006872852233675,
    'value': 3.26},
   'right': {'index': 'citric acid',
    'left': 6.017921146953405,
    'right': 5.548523206751055,
    'value': 0.25},
   'value': 0.3},
  'value': 0.48},
 'right': {'index': 'free sulfur dioxide',
  'left': {'index': 'fixed acidity',
   'left': {'index': 'pH',
```

Tree builted:

```
[XpH < 3.180]
 [Xsulphates < 0.480]
  [Xsulphates < 0.560]
   [Xsulphates < 0.630]
    [6.161016949152542]
    [6.095522388059702]
   [XpH < 3.270]
    [6.012383900928793]
    [5.761146496815287]
  [Xcitric acid < 0.300]
   [XpH < 3.260]
    [6.12536443148688]
    [5.8006872852233675]
   [Xcitric acid < 0.250]
    [6.017921146953405]
    [5.548523206751055]
 [Xfree sulfur dioxide < 34.000]
  [Xfixed acidity < 7.000]
   [XpH < 3.070]
    [5.962962962962963]
    [5.73482428115016]
   [Xsulphates < 0.480]
    [5.753289473684211]
    [5.718631178707224]
```

```
[Xfree sulfur dioxide < 22.000]
 [Xfree sulfur dioxide < 28.000]
  [6.053254437869822]
  [5.903846153846154]
 [Xfree sulfur dioxide < 15.000]
  [5.75]
  [5.32806324110672]
```

Predicted:

```
print(prediction);
5.73482428115016
6.161016949152542
6.161016949152542
5.73482428115016
5.32806324110672
5.73482428115016
6.161016949152542
5.32806324110672
5.32806324110672
5.73482428115016
5.548523206751055
5.32806324110672
5.32806324110672
5.32806324110672
6.161016949152542
6.161016949152542
5.32806324110672
5.73482428115016
5.73482428115016
```

For leaf node 0
Predicted value of the leaf node 5.73
Population in the leaf node 1731
MSE for this leaf 0.5774644643953113

For leaf node 1
Predicted value of the leaf node 6.16
Population in the leaf node 398
MSE for this leaf 0.6641751470922452

For leaf node 2
Predicted value of the leaf node 6.16
Population in the leaf node 398
MSE for this leaf 0.6993320875735459

For leaf node 3
Predicted value of the leaf node 5.73
Population in the leaf node 4
MSE for this leaf 0.6875

For leaf node 4
Predicted value of the leaf node 5.73
Population in the leaf node 1
MSE for this leaf 0.0

For leaf node 5

Predicted value of the leaf node 5.73
Population in the leaf node 1
MSE for this leaf 0.0

For leaf node 6
Predicted value of the leaf node 6.16
Population in the leaf node 2
MSE for this leaf 0.25

For leaf node 7
Predicted value of the leaf node 5.32
Population in the leaf node 3
MSE for this leaf 0.22222222222222224

For leaf node 8
Predicted value of the leaf node 5.73
Population in the leaf node 3
MSE for this leaf 0.22222222222222224

For leaf node 9
Predicted value of the leaf node 5.54
Population in the leaf node 2
MSE for this leaf 0.0

For leaf node 10
Predicted value of the leaf node 5.32
Population in the leaf node 1
MSE for this leaf 0.0

Explain how you decided the point at which to stop growing the regression tree.

- Initially I started with a near value of mean MSB I got.
- Then repeatedly run the code and changed the thresholds to get better results with less MSB.
- When I found the length of the decision tree is increasing more I stopped decreasing threshold value though the MSB slightly decreases with that.
- When the threshold value decreases to very less value or available data is very less, the correlation is giving NAN.

a. Show the structure of the resulting regression tree and the details of each leaf node including its population, predicted value and the MSE values for the training and the test datasets.

```
Out[927]: {'index': 'pH',
           'left': 5.960016155088853,
           'right': 5.793971924029727,
           'value': 3.18}
```

```
In [928]: # Print a decision tree
          def print_tree(node, depth=0):
              if isinstance(node, dict):
                  print('%s[X%s < %.3f]' % ((depth*' ', (node['index']), node['value'])))
                  print_tree(node['left'], depth+1)
                  print_tree(node['right'], depth+1)
              else:
                  print('%s[%s]' % ((depth*' ', node)))

          #print(tree)
          print_tree(tree)
```

```
[XpH < 3.180]
 [5.960016155088853]
 [5.793971924029727]
```

```
In [930]: test=X_test
          for i in range(len(test)):
              prediction = predict(tree, test.iloc[i])
              print(prediction);
```

```
5.793971924029727
5.960016155088853
5.960016155088853
5.793971924029727
5.793971924029727
5.793971924029727
5.960016155088853
5.793971924029727
5.793971924029727
5.793971924029727
5.960016155088853
5.793971924029727
5.793971924029727
5.793971924029727
5.960016155088853
5.960016155088853
5.793971924029727
5.793971924029727
5.793971924029727
```

b. Which paths are the best and the worst predictors of wine quality? Explain with numbers from the paths.

With lower threshold values:

```
[XpH < 3.180]
 [Xsulphates < 0.480]
  [Xsulphates < 0.560]
   [Xsulphates < 0.630]
    [6.155172413793103]
    [6.097859327217125]
   [XpH < 3.270]
    [6.006410256410256]
    [5.764900662251655]
  [Xcitric acid < 0.300]
   [XpH < 3.260]
    [6.119631901840491]
    [5.795698924731183]
   [Xcitric acid < 0.250]
    [6.025547445255475]
    [5.556521739130435]
 [Xfree sulfur dioxide < 34.000]
  [Xfixed acidity < 7.000]
   [XpH < 3.070]
    [5.964497041420119]
    [5.754152823920266]
   [Xsulphates < 0.480]
    [5.752475247524752]
    [5.719844357976654]
  [Xfree sulfur dioxide < 22.000]
   [Xfree sulfur dioxide < 28.000]
    [6.057228915662651]
    [5.908496732026144]
```

- As per my knowledge, the path which is shorted gives the results with less number of iterations which can be considered as best path.
- A path with more nodes/ deepest branches is considered as worth path because in this to take a decision need to go with more number of iterations.
- When I changed the threshold the graph size increased. In the above graph everything is similar distance from the root node, so as per my knowledge everything is a worst and best path.

```
In [917]: Count, MSB_list

Out[917]: ([(2422, 2476),
            (1150, 1326),
            (637, 689),
            (335, 354),
            (173, 181),
            (81, 100),
            (41, 59),
            (29, 30),
            (14, 16),
            (14, 15),
            (6, 8),
            (4, 4),
            (1, 3),
            (2422, 2476),
            (2422, 2476)],
           [(166.59357099630026, 187.13524660648275),
            (186.82760151563488, 136.69764887610157),
            (194.21573811745398, 178.4143416714303),
            (195.3338033928552, 191.33112483422065),
            (139.02008970433022, 253.30924496646892),
            (135.16207085335014, 141.7878885333419),
            (152.39404411380903, 102.5705127156154),
            (137.6097093052717, 104.46741424808948),
            (94.82266582182034, 45.09734455150365),
            (63.02124851653725, 147.03896644243943),
            (169.95007650998699, 109.9323939127528),
            (80.22447908786458, 185.13758747687498),
            (230.24231363870368, 0.0),
            (166.59357099630026, 187.13524660648275),
            (166.59357099630026, 187.13524660648275)])
```

4. (8) Use the red wine dataset as the test set on the regression tree obtained for white wines in #3 above and compute the MSE for the records of the red wine dataset. Compare this performance to that for the white wine. What can you conclude?

```
In [932]: test=X_test
          for i in range(len(test)):
              prediction = predict(tree, test.iloc[i])
              print(prediction);

          5.793971924029727
          5.793971924029727
          5.793971924029727
          5.960016155088853
          5.793971924029727
          5.793971924029727
          5.793971924029727
          5.793971924029727
          5.793971924029727
          5.793971924029727
          5.793971924029727
          5.793971924029727
          5.793971924029727
          5.793971924029727
          5.960016155088853
          5.960016155088853
          5.793971924029727
          5.960016155088853
```

```
In [938]: Count, MSB_list

Out[938]: ([(2422, 2476)], [(166.59357099630026, 187.13524660648275)])
```

Comparison with Red and Wight wine results:

For red wine the MSB results are more than the Wight wine data set. May be because the red wine data might be different than the Wight wine.

In prediction also, by observation I found over all good accuracy.

5. Consider the following transactions: (T1: A B C E G H), (T2: A B E F M), (T3: B C D E G M), (T4: A B C H), (T5: C D E F M), (T6: A B C E H), (T7: B C E G H M). Perform the following:
   a. (8) Show all steps towards building the FP-Growth Tree for these transactions.
   b. (8) Show the steps for2 finding the frequent itemsets (min. Support 3) from the FP-Growth tree and stop after finding the first four frequent itemsets. Briefly describe each step.
   c. (8) Show the execution of GenMax algorithm with this dataset and stop after finding the first three Maximal Frequent Itemsets. Briefly describe each step.
   d. (8) Show the execution of the CHARM algorithm with this dataset and stop after finding the first three closed itemsets. Briefly describe each step.
6. (5) These five extra bonus points are for well-organized and well-presented answers to the above questions in your submission.

$T_1$ : A B C E G H

$T_2$ : A B E F M

$T_3$ : B C D E G M

$T_4$ : A B C H

$T_5$ : C D E F M

$T_6$ : A B C E H

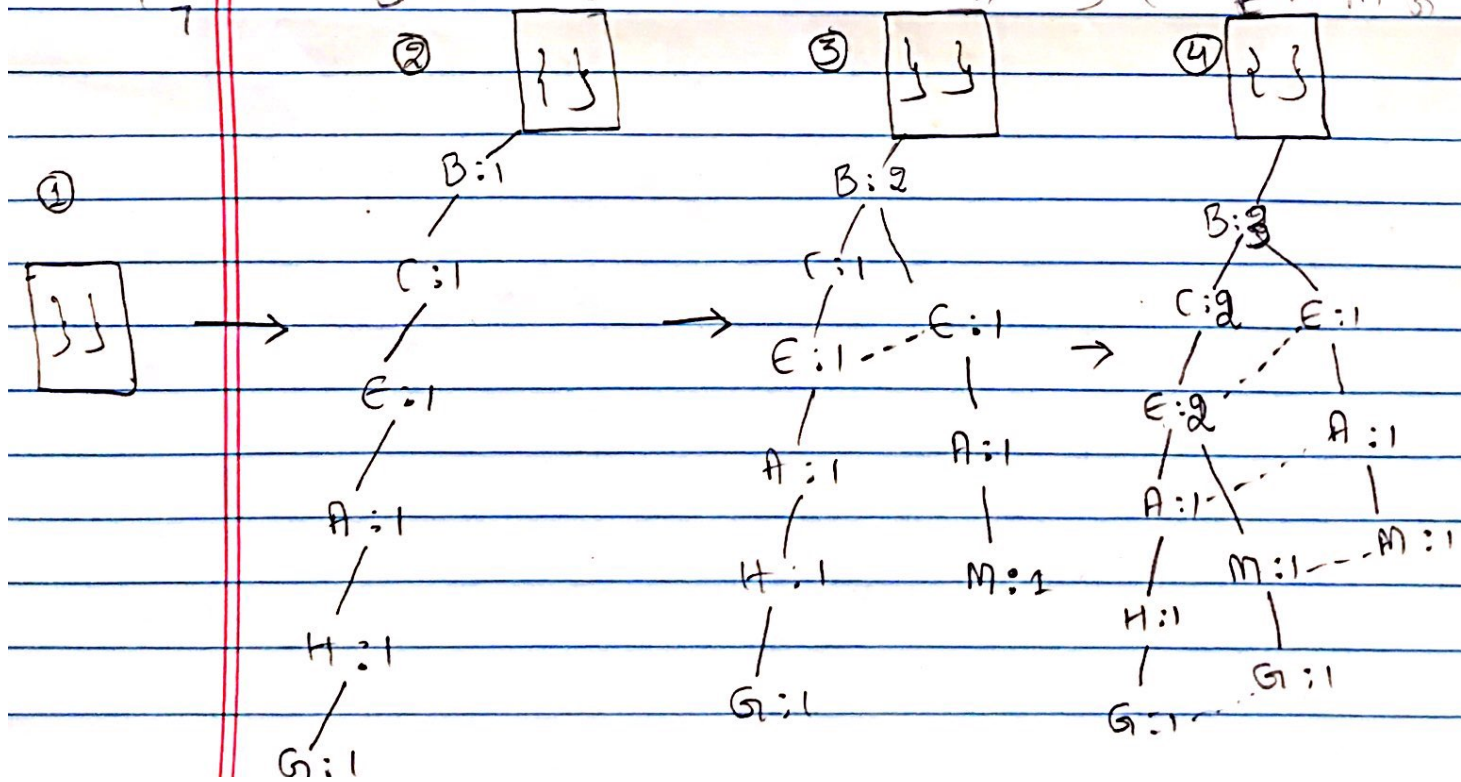
$T_7$ : B C E G H M

L

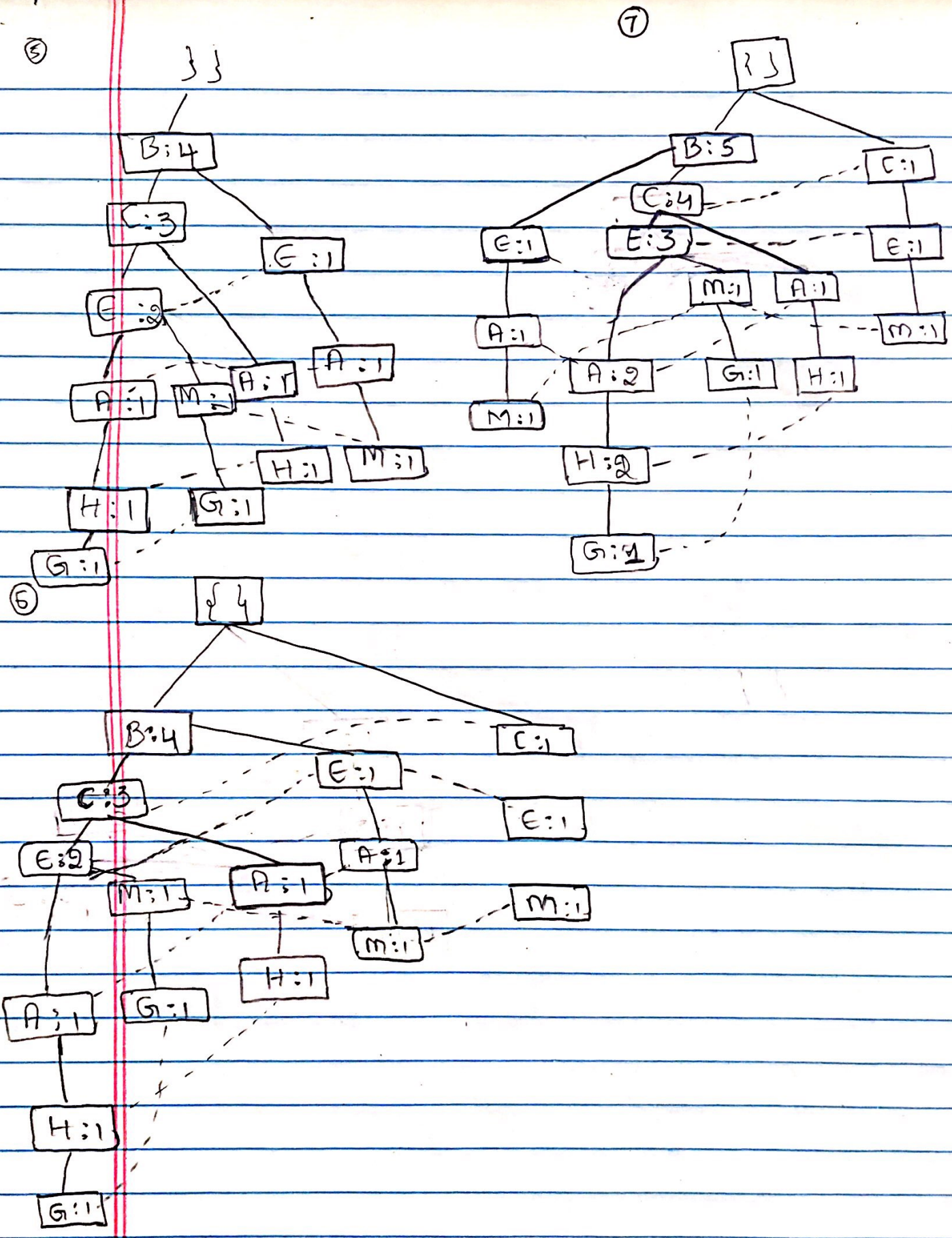
| Item | L = |
|---|---|
| A - 4 - | B - 6 |
| B - 6 - | C - 6 |
| C - 6 | E - 6 |
| D - 2 | A - 4 |
| E - 6 - | H - 4 |
| F - 2 | M - 4 |
| G - 3 | G - 3 |
| H - 4 - | |
| M - 4 - | |

order each frequent items

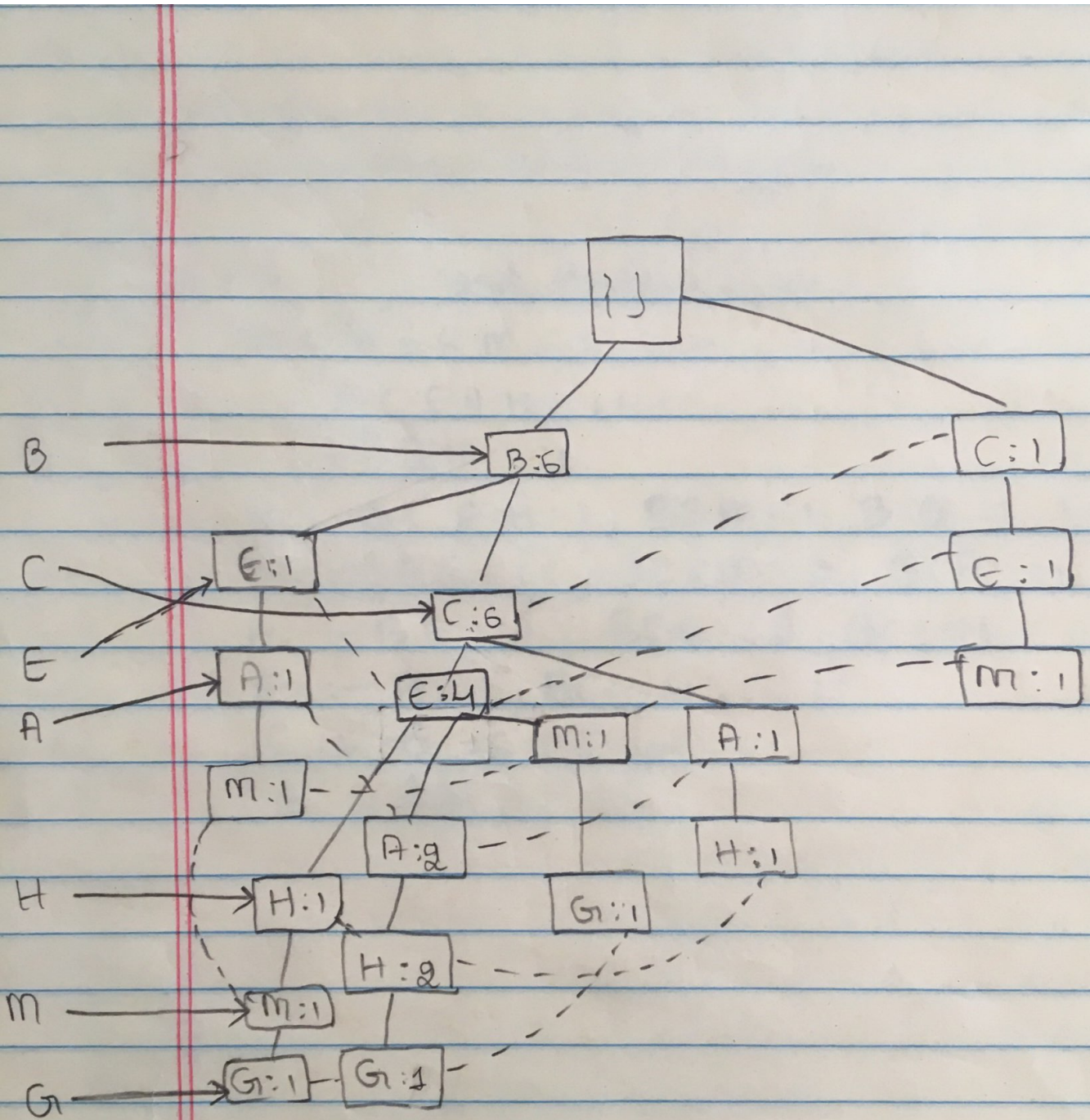
| | | |
|---|---|---|
| $T_1$ | A B C E G H | 2) B C E A H G |
| $T_2$ | A B E F M | 3) B E A M |
| $T_3$ | B C D E G M | 4) B C E M G |
| $T_4$ | A B C H | 5) B C A H |
| $T_5$ | C D E F M | 6) C E M |
| $T_6$ | A B C E H | 7) B C E A H |
| $T_7$ | B C E G H M | 8) B C E H M G |




Scanned by CamScanner




Scanned by CamScanner




Scanned by CamScanner

Step-1 :-

conditional pattern bases

| item | cond. Pattern base. |
|---|---|
| G | :- BCEHM :1 |
| | BCEAH :1 |
| | BCEM :1 |
| M | :- BCEH :1, BEA :1, BCE :1, CE :1 |
| H | :- BCE :1, BCEA :2, BCA :1 |
| A | :- BE :1, BCE :2, BC :1 |
| E | :- B :1, BC :4, C :1 |
| C | :- B :6 |
| B | :- {} |

Scanned by CamScanner

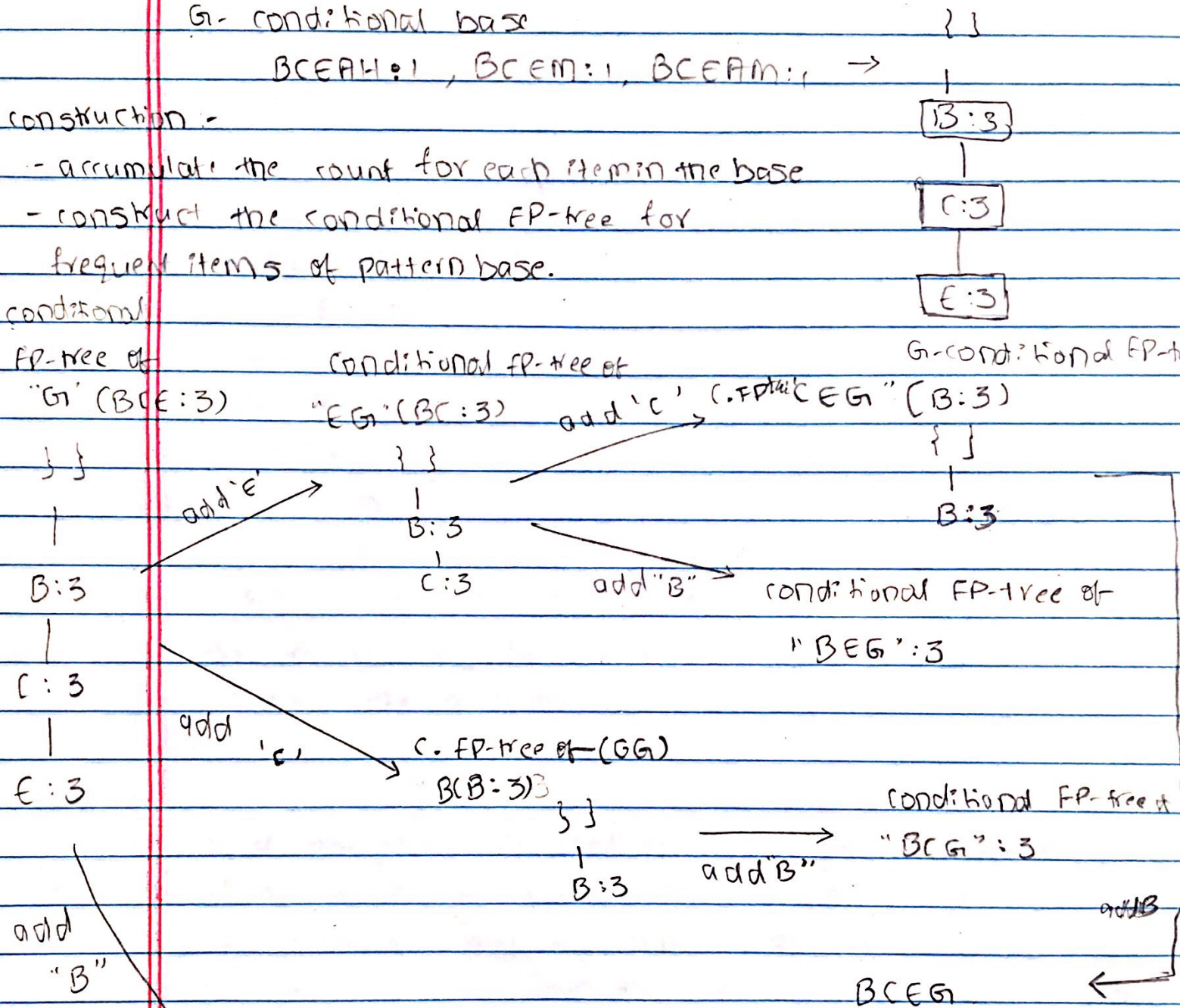
construct conditional FP-Tree

G- conditional base

BCEAH:1 , BCEM:1, BCEAM:1 →

construction -

- accumulate the count for each item in the base
- construct the conditional FP-tree for frequent items of pattern base.




frequent patterns :-

G, EG, B, CEG, BEG, BCG, BCEG

Scanned by CamScanner

c) GenMax Algorithm.

Given:

$T_1$ : A B C E G H

$T_2$ : A B E F M

$T_3$ : B C D E G M

$T_4$ : A B C H

$T_5$ : C D E F M

$T_6$ : A B C E H

$T_7$ : B C E G H M

Step:-

| A | B | C | D | E | F | G | H | M |
|---|---|---|---|---|---|---|---|---|
| 1 2 4 6 | 1 2 3 4 6 7 | 1 3 4 5 6 7 | 3 5 | 1 2 3 5 6 7 | 2 5 | 1 3 7 | 1 4 6 7 | 2 3 5 7 |

1, write the transactions for every item set this will be the starting ip for the algorithm.

Step 2:-

Check if the entire current branch can be pruned by checking if the union of all the itemsets. $Y = \{x_i\}$ is already subsumed by some maximal pattern $Z \in M$. If so, no maximal itemset can be generated from the current branch and it is pruned. On the other hand, if the branch is not pruned intersect each pair $(x_i, t(x_i))$ with all other $(x_j, t(x_j))$ with $j > i$ to generate new candidates $X_{ij}$ which are added to pairset $P_i$.

repeat this until all maximal frequency sets are found

Scanned by CamScanner

| A | B | C | D | E | F | G | H | M |
|---|---|---|---|---|---|---|---|---|
| 1246 | 123467 | 134567 | 35 | 123567 | 25 | 137 | 1467 | 235 |

$P_A$

$P_B$

| AB | AC | AE |
|---|---|---|
| 1246 | 1346 | 126 |

| BE | BE | BG | BH |
|---|---|---|---|
| 13467 | 12367 | 137 | 1467 |

$P_{AB}$

$P_{BC}$

| ABC | ABE | ABH |
|---|---|---|
| 146 | 26 | 146 |

| BCE | BCG | BCH |
|---|---|---|
| 1367 | 137 | 1467 |

$P_{ABC}$

$P_{BCE}$

| ABCH |
|---|
| 146 |

| BCEG | BCEH |
|---|---|
| 137 | 167 |

Frequency item sets:

ABCH, BCEG, BCEH

Scanned by CamScanner

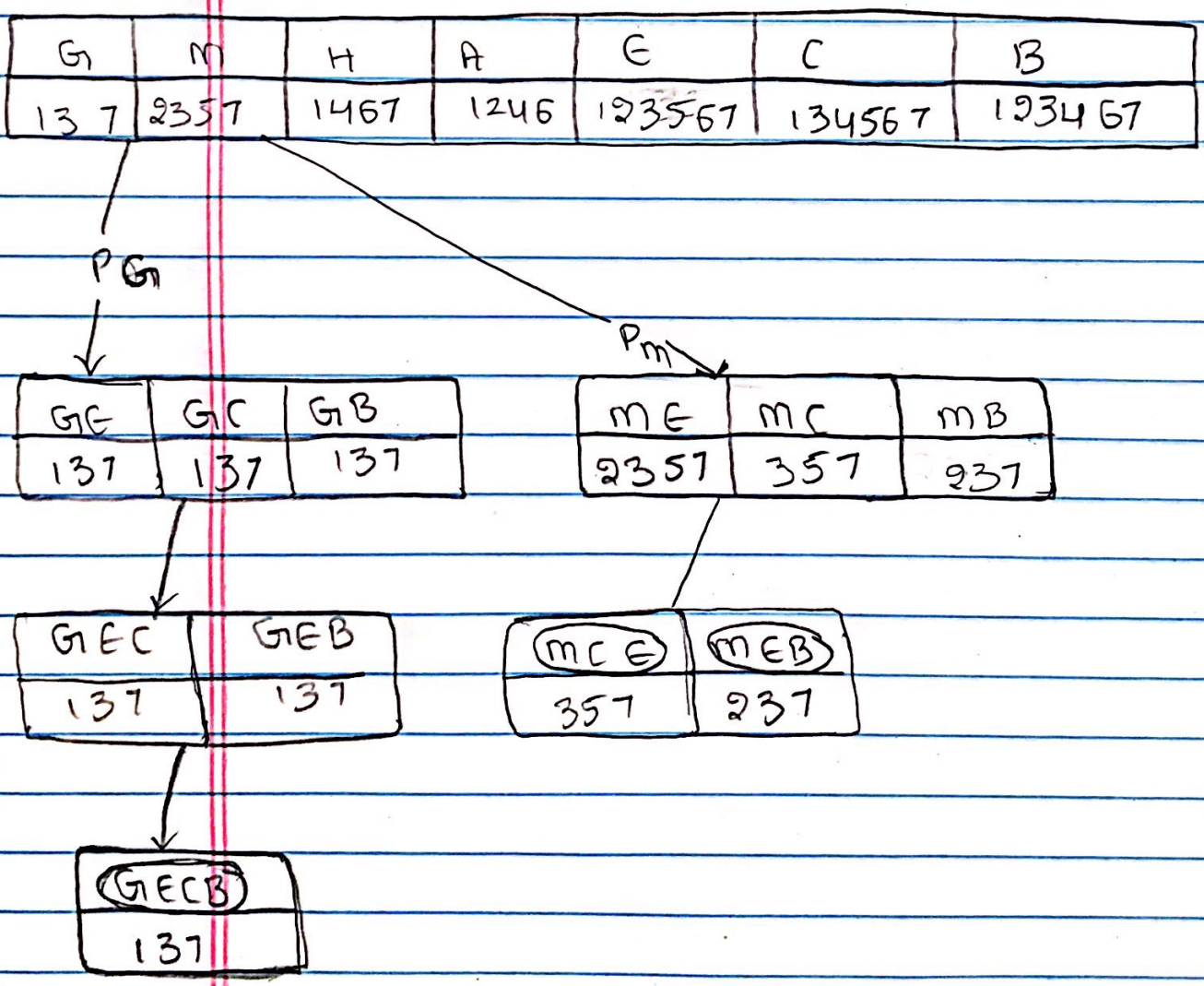
(d)

Charm algorithm:-

Step1:- Sort P( initial set of IT-pairs) in increasing order of support.

Step2:- process extention from A. prune the frequent. continuously until we get all closed item sets containing A has been found.

step3:- continue charming algorithm for rest of branches.




Scanned by CamScanner

Full regression tree for 2 problem:

```
[XFD_Worst < 0.087]
 [XFD_SE < 0.342]
  [XFD_SE < 0.392]
   [XPerimeter_Worst < 0.231]
    [XSymmetry_Mean < 0.153]
     [XSymmetry_Mean < 0.159]
      [XTexture_SE < 0.123]
       [XTexture_SE < 0.135]
        [123.0]
        [119.0]
       [XRadius_SE < 15.945]
        [66.0]
        [61.0]
      [XTumor_Size < 3.000]
       [XRadius_SE < 15.945]
        [116.0]
        [97.0]
       [58.0]
     [XTexture_Mean < 904.200]
      [XTexture_Mean < 1311.000]
       [XRadius_Mean < 21.230]
        [116.0]
        [53.0]
       [9.0]
      [XCV_Mean < 26.380]
       [XRadius_Mean < 13.010]
        [86.0]
        [43.0]
       [20.0]
    [XFD_SE < 0.427]
     [XTumor_Size < 2.750]
      [XSmoothness_SE < 31.000]
       [XTexture_SE < 0.103]
        [116.0]
        [73.0]
       [76.0]
      [XSymmetry_Worst < 0.497]
       [XRadius_Mean < 15.290]
        [109.0]
        [72.0]
       [38.0]
     [XFD_SE < 0.412]
      [XPerimeter_SE < 0.074]
       [XRadius_Mean < 14.560]
        [103.0]
        [36.0]
       [7.0]
```

```
   [XSymmetry_Mean < 0.146]
    [XArea_Mean < 0.064]
     [52.0]
     [27.0]
    [30.0]
[XFD_SE < 0.365]
 [XFD_SE < 0.371]
  [XFD_SE < 0.379]
   [XArea_Worst < 0.907]
    [XArea_Worst < 0.953]
     [125.0]
     [58.0]
    [XTexture_SE < 0.113]
     [111.0]
     [8.0]
   [XTumor_Size < 2.600]
    [XConcavity_Mean < 0.017]
     [114.0]
     [10.0]
    [3.0]
  [XLymph_Node_Status < 0.000]
   [XLymph_Node_Status < 0.000]
    [XLymph_Node_Status < 0.000]
     [XLymph_Node_Status < 0.000]
      [XLymph_Node_Status < 0.000]
       [39.833333333333336]
       [nan]
      [nan]
     [nan]
    [nan]
   [nan]
 [XSmoothness_Worst < 0.006]
  [XTumor_Size < 2.000]
   [XFD_Mean < 0.171]
    [XRadius_SE < 18.335]
     [96.0]
     [77.0]
    [73.0]
   [XFD_SE < 0.358]
    [XRadius_Mean < 15.605]
     [68.0]
     [44.0]
    [3.0]
  [XCompactness_Worst < 0.012]
   [XConcavity_Mean < 0.019]
    [XArea_SE < 0.648]
     [62.0]
     [35.0]
    [8.0]
   [XTumor_Size < 2.500]
    [XArea_SE < 0.264]
```

```
      [31.0]
      [27.0]
     [17.0]
[XSymmetry_Mean < 0.151]
 [XTexture_SE < 0.112]
  [XArea_Mean < 0.069]
   [XFD_SE < 0.320]
    [XCV_Worst < 1015.400]
     [XRadius_Mean < 15.620]
      [104.0]
      [98.0]
     [XRadius_SE < 19.050]
      [77.0]
      [56.0]
    [XFD_SE < 0.308]
     [XTexture_SE < 0.130]
      [101.0]
      [91.0]
     [41.0]
   [XTexture_SE < 0.114]
    [XArea_Mean < 0.065]
     [XRadius_Mean < 12.865]
      [106.0]
      [99.0]
     [XArea_Mean < 0.063]
      [63.0]
      [5.0]
    [XRadius_Mean < 14.090]
     [49.0]
     [19.0]
  [XFD_Worst < 0.106]
   [XFD_SE < 0.303]
    [XArea_Mean < 0.069]
     [XRadius_Mean < 12.585]
      [77.0]
      [74.0]
     [70.0]
    [XFD_Mean < 0.179]
     [XSymmetry_Mean < 0.157]
      [76.0]
      [68.0]
     [34.0]
   [XSymmetry_Mean < 0.159]
    [XPerimeter_Worst < 0.167]
     [XRadius_Mean < 16.570]
      [55.0]
      [9.0]
     [5.0]
    [XCompactness_Worst < 0.013]
     [XTexture_Mean < 530.050]
      [17.0]
```

```
     [15.0]
    [14.0]
[XSmoothness_Worst < 0.006]
 [XArea_Worst < 1.208]
  [XSmoothness_Worst < 0.008]
   [XCV_Mean < 30.860]
    [XLymph_Node_Status < 0.500]
     [105.0]
     [64.0]
    [34.0]
   [XSmoothness_Mean < 5.801]
    [XRadius_Mean < 18.350]
     [40.0]
     [32.0]
    [11.0]
  [XTumor_Size < 3.000]
   [XConcavity_Mean < 0.018]
    [XTexture_SE < 0.099]
     [75.0]
     [19.0]
    [XArea_Mean < 0.066]
     [38.0]
     [3.0]
   [XRadius_Mean < 18.585]
    [17.0]
    [8.0]
 [XFD_SE < 0.290]
  [XFD_Worst < 0.094]
   [XCV_Mean < 26.930]
    [XRadius_SE < 22.005]
     [81.0]
     [39.0]
    [58.0]
   [XSymmetry_Mean < 0.140]
    [XRadius_Mean < 18.600]
     [56.0]
     [16.0]
    [5.0]
  [XTumor_Size < 2.000]
   [XTexture_SE < 0.101]
    [XRadius_Mean < 16.845]
     [41.0]
     [26.0]
    [5.0]
   [XTumor_Size < 1.000]
    [XTumor_Size < 1.000]
     [XTumor_Size < 1.000]
      [XTumor_Size < 1.000]
       [26.0]
       [nan]
      [nan]
```

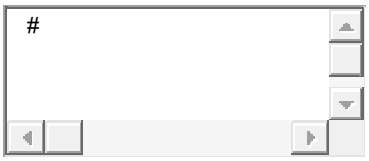
```
       [nan]
      [nan]
[XFD_Worst < 0.077]
 [XFD_Worst < 0.081]
  [XSmoothness_Worst < 0.005]
   [XArea_Mean < 0.061]
    [XTumor_Size < 1.500]
     [XConcavity_SE < 0.003]
      [XArea_Worst < 1.332]
       [XRadius_Mean < 17.385]
        [108.0]
        [100.0]
       [91.0]
      [XRadius_SE < 23.840]
       [83.0]
       [78.0]
     [XRadius_SE < 22.975]
      [87.0]
      [47.0]
    [XCV_Mean < 28.830]
     [XArea_Worst < 1.199]
      [XArea_Worst < 1.220]
       [123.0]
       [88.0]
      [12.0]
     [XTexture_Worst < 0.130]
      [XTexture_SE < 0.099]
       [57.0]
       [48.0]
      [11.0]
   [XCV_Mean < 32.835]
    [XFD_Worst < 0.083]
     [XConcavity_SE < 0.003]
      [XPerimeter_Worst < 0.149]
       [123.0]
       [67.0]
      [31.0]
     [XCV_Mean < 37.080]
      [XRadius_Mean < 19.375]
       [63.0]
       [31.0]
      [17.0]
    [XSymmetry_Mean < 0.141]
     [XCompactness_Worst < 0.011]
      [XTexture_SE < 0.099]
       [76.0]
       [70.0]
      [65.0]
     [XSmoothness_Worst < 0.004]
      [XCompactness_SE < 0.029]
       [23.0]
```

```
     [17.0]
    [2.0]
[XArea_Mean < 0.059]
 [XTumor_Size < 2.500]
  [XTumor_Size < 3.500]
   [XTumor_Size < 3.750]
    [XArea_Mean < 0.061]
     [XFD_Worst < 0.078]
      [94.0]
      [44.0]
     [1.0]
    [XCompactness_Worst < 0.018]
     [XCompactness_Worst < 0.019]
      [27.0]
      [14.0]
     [6.0]
   [17.0]
  [XTexture_SE < 0.100]
   [XFD_Worst < 0.080]
    [XRadius_Mean < 16.470]
     [65.0]
     [58.0]
    [40.0]
   [XFD_SE < 0.291]
    [XRadius_Mean < 19.865]
     [24.0]
     [8.0]
    [10.0]
 [XFD_Worst < 0.079]
  [XCV_Mean < 34.675]
   [XArea_Mean < 0.055]
    [XArea_SE < 0.409]
     [59.0]
     [39.0]
    [13.0]
   [XRadius_SE < 21.680]
    [XRadius_SE < 21.850]
     [48.0]
     [33.0]
    [7.0]
  [XSmoothness_SE < 57.340]
   [XCV_Worst < 1816.000]
    [XRadius_Mean < 22.140]
     [24.0]
     [10.0]
    [8.0]
   [XFD_Worst < 0.078]
    [XRadius_Mean < 16.190]
     [16.0]
     [13.0]
    [1.0]
```

```
[XFD_SE < 0.271]
 [XLymph_Node_Status < 1.000]
  [XSymmetry_Worst < 0.290]
   [XLymph_Node_Status < 3.500]
    [XLymph_Node_Status < 7.000]
     [XRadius_SE < 18.240]
      [73.0]
      [36.0]
     [XRadius_Mean < 16.425]
      [37.0]
      [15.0]
    [XLymph_Node_Status < 1.500]
     [XTumor_Size < 4.250]
      [91.0]
      [51.0]
     [XRadius_SE < 23.270]
      [62.0]
      [10.0]
   [XSymmetry_Worst < 0.231]
    [XSymmetry_Worst < 0.258]
     [XTexture_SE < 0.085]
      [84.0]
      [66.0]
     [XRadius_Mean < 17.395]
      [12.0]
      [5.0]
    [XLymph_Node_Status < 1.500]
     [XRadius_SE < 28.300]
      [51.0]
      [13.0]
     [XRadius_Mean < 17.620]
      [35.0]
      [12.0]
  [XFD_Worst < 0.074]
   [XFD_Worst < 0.076]
    [XTexture_Worst < 0.113]
     [XTexture_SE < 0.092]
      [104.0]
      [74.0]
     [36.0]
    [XCompactness_Mean < 0.047]
     [29.0]
     [9.0]
   [XSmoothness_Worst < 0.008]
    [XSmoothness_Worst < 0.009]
     [117.0]
     [11.0]
    [XArea_Mean < 0.055]
     [34.0]
     [12.0]
 [XCompactness_Mean < 0.019]
```

```
[XArea_Mean < 0.057]
 [XArea_Mean < 0.058]
  [XArea_Mean < 0.059]
   [XRadius_Mean < 19.720]
    [84.0]
    [13.0]
   [74.0]
  [XArea_SE < 0.853]
   [XRadius_SE < 20.840]
    [55.0]
    [16.0]
   [6.0]
 [XTexture_Worst < 0.108]
  [XFD_SE < 0.227]
   [XPerimeter_Worst < 0.159]
    [61.0]
    [34.0]
   [28.0]
  [XFD_Worst < 0.061]
   [XRadius_Mean < 19.860]
    [20.0]
    [9.0]
   [10.0]
[XSmoothness_Worst < 0.005]
 [XRadius_Mean < 14.795]
  [XLymph_Node_Status < 0.000]
   [XLymph_Node_Status < 0.000]
    [XLymph_Node_Status < 0.000]
     [XLymph_Node_Status < 0.000]
      [53.333333333333336]
      [nan]
     [nan]
    [nan]
   [nan]
  [XFD_SE < 0.218]
   [XRadius_Mean < 14.230]
    [57.0]
    [26.0]
   [1.0]
 [XArea_Mean < 0.055]
  [XArea_Mean < 0.056]
   [XPerimeter_Worst < 0.149]
    [38.0]
    [14.0]
   [1.0]
  [XCV_Mean < 23.360]
   [XRadius_SE < 19.060]
    [62.0]
    [7.0]
   [12.0]
```

In [ ]:




*# Make a prediction with a deci*

# coding: utf-8

# In[794]:

```
from sklearn.model_selection import KFold
import numpy as np
from sklearn.model_selection import train_test_split
import pydotplus
import pandas as pd
from sklearn.externals.six import StringIO
from sklearn.tree import export_graphviz
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import train_test_split
import pydotplus
import pandas as pd
from sklearn.externals.six import StringIO
from sklearn.tree import export_graphviz
from sklearn.tree import DecisionTreeClassifier
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.tree import DecisionTreeClassifier
from sklearn.tree import export_graphviz
from sklearn.grid_search import GridSearchCV
from sklearn.grid_search import RandomizedSearchCV
from sklearn import tree
import pydotplus
from sklearn.cross_validation import  cross_val_score
from IPython.display import Image
from sklearn.externals.six import StringIO
from sklearn.utils import shuffle
from sklearn.metrics import precision_recall_fscore_support
from sklearn.metrics import mean_squared_error
import statistics as st
```

# In[897]:

```
#read CSV files of 2 data sets
my_data = pd.read_csv("C:/Users/mereddda/Desktop/IDA/breast-cancer-wisconsin.csv",
skipinitialspace=True)
my_data.drop(my_data.columns[[1]], axis=1, inplace=True)
my_data.drop(my_data.columns[[0]], axis=1, inplace=True)
my_data.head()
```

# In[898]:

```
my_data.head()
X=my_data.iloc[:,1:]
Y=my_data.iloc[:,0]

X_test=X.iloc[:132,:]
X_train=X.iloc[132:,:]
Y_test=Y.iloc[:132]
Y_test
Y_train=Y.iloc[132:]
#Y_train
#X_train
```

# In[918]:

```
#wine:
#read CSV files of 2 data sets
my_data = pd.read_csv("C:/Users/mereddda/Desktop/IDA/winequality-white.csv",
skipinitialspace=True)
my_data.head()
train,test=train_test_split(my_data,random_state=20, test_size=0.33)
X_train=train.iloc[:,:-2]
Y_train=train.iloc[:,-1]
X_test=test.iloc[:,:-2]
Y_test=test.iloc[:,-1]
```

# In[931]:

```
my_data = pd.read_csv("C:/Users/mereddda/Desktop/IDA/winequality-red.csv",
skipinitialspace=True)
X_test=my_data.iloc[:,:-2]
Y_test=my_data.iloc[:,-1]
```

# In[919]:

```
# Split a dataset based on an attribute and an attribute value
def test_split(index, value, dataset):
        left, right = list(), list()
        for row in dataset:
                if row[index] < value:
                        left.append(row)
                else:
                        right.append(row)
        return left, right
```

# In[920]:

```
def df_split(df, feature, value):
 left = df[df[feature]<=value]
 right = df[df[feature]>value]
 return left, right #
```

# In[921]:

```
global Count
Count=list()
global MSB_list
MSB_list=list()
# Select the best split point for a dataset
def get_split(dataset):
#    X=dataset.iloc[:,:-2]
#    Y=dataset.iloc[:,-1]
```

```
    X=dataset.iloc[:,:-2]
    Y=dataset.iloc[:,-1]
    b_index, b_value, b_score, b_groups = 999, 999, 999, None
    print(X);print(Y)
    correlation =X.apply(lambda x: x.corr(Y))
    print('correlation'); print(correlation)
    max_col = correlation.idxmax()
    print(max_col)

    col_med = st.median(dataset.loc[:,max_col])
    right = dataset.loc[dataset.loc[:,max_col] >= col_med]
    left = dataset.loc[dataset.loc[:,max_col] < col_med]
#    groups=df_split(dataset, max_col, col_med)
#    print(type(groups))
#    tup=(4,5)
#    print(type(tup))
    b_groups=(right,left)
#    dataset = dataset.drop(max_col,1)
#     return left, right #(left df, right df
    c=(len(left),len(right))
    Count.append(c)

#    msb=(MSE_error(left),MSE_error(right))
#    Count.append(msb)
    return {'index':max_col, 'value':col_med, 'groups':b_groups}
#get_split(my_data)


# In[922]:


# Create a terminal node value
def to_terminal(group):
    outcomes = group.loc[:,'quality'].mean()
    return outcomes


# In[923]:


# Create child splits for a node or make terminal
def split(node):
 left, right = node['groups']
 del(node['groups'])
```

```
 # check for a no split
 if len(left) == 0 or len(right) == 0 :
   return
 # process left child
 left_MSE = MSE_error(left)
 right_error = MSE_error(right)
 msb=(left_MSE,right_error)
 MSB_list.append(msb)
 if left_MSE <= 500:
   node['left'] = to_terminal(left)
 else:
   node['left'] = get_split(left)
   split(node['left'])
 # process right child
 if right_error <= 500:
   node['right'] = to_terminal(right)
 else:
   node['right'] = get_split(right)
   split(node['right'])


# In[840]:


# Create child splits for a node or make terminal
def split(node, max_depth=4, min_size=1, depth=1):
        left, right = node['groups']
        del(node['groups'])
        # check for a no split
#       if not left or not right:
#               node['left'] = node['right'] = to_terminal(left + right)
#               return
        # check for max depth
        if depth >= max_depth:
                node['left'], node['right'] = to_terminal(left), to_terminal(right)
                return
        # process left child
        if len(left) <= min_size:
                node['left'] = to_terminal(left)
        else:
                node['left'] = get_split(left)
                split(node['left'], max_depth, min_size, depth+1)
        # process right child
        if len(right) <= min_size:
```

```
                node['right'] = to_terminal(right)
        else:
                node['right'] = get_split(right)
                split(node['right'], max_depth, min_size, depth+1)


# In[924]:


def build_tree(train):
        root = get_split(train) # best split
        split(root) # split add to func
        return root


# In[925]:


test.head()


# In[926]:


# Print a decision tree
def print_tree(node, depth=0):
        if isinstance(node, dict):
                print ((depth, (node['index']+1), node['value']))
                print_tree(node['left'], depth+1)
                print_tree(node['right'], depth+1)
        else:
                print('%s[%s]' % ((depth*' ', node)))


# In[927]:


#my_data.drop(my_data.columns[[0]], axis=1, inplace=True)
my_data.head()
tree = build_tree(my_data)
tree


# In[928]:
```

```
# Print a decision tree
def print_tree(node, depth=0):
	if isinstance(node, dict):
		print('%s[X%s < %.3f]' % ((depth*' ', (node['index']), node['value'])))
		print_tree(node['left'], depth+1)
		print_tree(node['right'], depth+1)
	else:
		print('%s[%s]' % ((depth*' ', node)))

#print(tree)
print_tree(tree)


# In[929]:


# Make a prediction with a decision tree

def predict(node, row):
#	print(node['right'])
	if row[node['index']] < node['value']:
		if isinstance(node['left'], dict):
			return predict(node['left'], row)
		else:
			return node['left']

	else:
		if isinstance(node['right'], dict):
			return predict(node['right'], row)
		else:
			return node['right']


# In[937]:


test=X_test
for i in range(len(test)):
	prediction = predict(tree, test.iloc[i])
	print(prediction);
```

# In[938]:

Count, MSB_list

# In[689]:

```
dataset = [[2.771244718,1.784783929,0],
        [1.728571309,1.169761413,0],
        [3.678319846,2.81281357,0],
        [3.961043357,2.61995032,0],
        [2.999208922,2.209014212,0],
        [7.497545867,3.162953546,1],
        [9.00220326,3.339047188,1],
        [7.444542326,0.476683375,1],
        [10.12493903,3.234550982,1],
        [6.642287351,3.319983761,1]]

#  predict with a stump
stump = {'index': 0, 'right': 1, 'value': 6.642287351, 'left': 0}
for row in dataset:
        prediction = predict(stump, row)
        print(row)
        print('Expected=%d, Got=%d' % (row[-1], prediction))
```

# In[690]:

```
y_predict = [y_pred] * len(Y_train)
train_MSE_error = mean_squared_error(Y_train, y_predict)
train_MSE_error
```

# In[691]:

```
y_predict = [y_pred] * len(Y_test)
test_MSE_error  = mean_squared_error(Y_test, y_predict)
test_MSE_error
```

# In[692]:

```
X_train_sample = X_train;
```

# In[693]:

```
correlation = X_train_sample.apply(lambda x: x.corr(Y_train))
max_col = correlation.idxmax()
col_med = st.median(X_train.loc[:,max_col])
print(col_med)
datax = np.where(X_train.loc[:,max_col] > col_med)
datax
```

# In[694]:

```
X_train_right = X_train.loc[X_train.loc[:,max_col] >= col_med]
X_train_left = X_train.loc[X_train.loc[:,max_col] < col_med]
X_train_sample = X_train_sample.drop(max_col,1)
X_train_sample
```

# In[695]:

```
def database_split(df):
    correlation =df.apply(lambda x: x.corr(Y_train))
    max_col = correlation.idxmax()
    col_med = st.median(df.loc[:,max_col])
    right = df.loc[df.loc[:,max_col] >= col_med]
    left = df.loc[df.loc[:,max_col] < col_med]
    df = df.drop(max_col,1)
    return left, right #(left df, right df
database_split(X_train)
```

# In[696]:

```
def MSE_error(df):
```

```
    y_pred = df.mean()
    y_predict = [y_pred] * len(df)
    train_MSE_error = mean_squared_error(df, y_predict)
    return train_MSE_error
MSE_error(Y_train)


# In[697]:


left,right=database_split(X_train)
#database_split(Y_train)
if(database_split())


# In[698]:


right
```