Decision Tree

Problem 2

CART Algorithm

Sample For Original data

	id	age	gender	height	weight	ap_hi	ap_lo	cholesterol	gluc	smoke	alco	active	cardio
0	0	18393	2	168	62.0	110	80	1	1	0	0	1	0
1	1	20228	1	156	85.0	140	90	3	1	0	0	1	1
2	2	18857	1	165	64.0	130	70	3	1	0	0	0	1
3	3	17623	2	169	82.0	150	100	1	1	0	0	1	1
4	4	17474	1	156	56.0	100	60	1	1	0	0	0	0

I do some processing on data features

- **The age is given in data so I convert it to years**
- **I don't use id column so I dropped it**
- **Check on duplicates in data and I found 3208 duplicates**

```
[7]  M↓
     data.duplicated().sum()

     3208
```

- **Check on outliers That is**
    1. **systolic_bp feature >200 or < 80**
    2. **diastolic_bp feature > 180 or <50**
- **Drop outliers**
- **I Thought Height and weight seems uncorrelated with cardio feature But BMI (Body Mass Index) could be more helpful so I replace two columns height and weight with one column called BMI**
- **To make all features take only discrete value I tried to categorize each feature**
    1. **I Found For BMI feature**
       BMI between 18.5 and 25 , person Normal
       if BMI obove 25 , person is obese

if BMI less than 18.5 , person is underweight
So I categorized it with 0 , 1 , 2

2. I Found for systolic blood pressure number
   Normal: Below 120
   Elevated: 120-129
   Stage 1 high blood pressure (also called hypertension): 130-139
   Stage 2 hypertension: 140 or more
   Hypertensive crisis: 180 or more
   So I categorized it with 0 , 1 , 2 ,3 ,4

3. I Found for diastolic blood pressure number
   Normal: Lower than 80
   Stage 1 hypertension: 80-89
   Stage 2 hypertension: 90 or more*
   Hypertensive crisis: 120 or more
   So I categorized it with 0 , 1 , 2 ,3

4. I found most of unique values in age in range 30 to 60
   So I categorized it to 5 Catagories

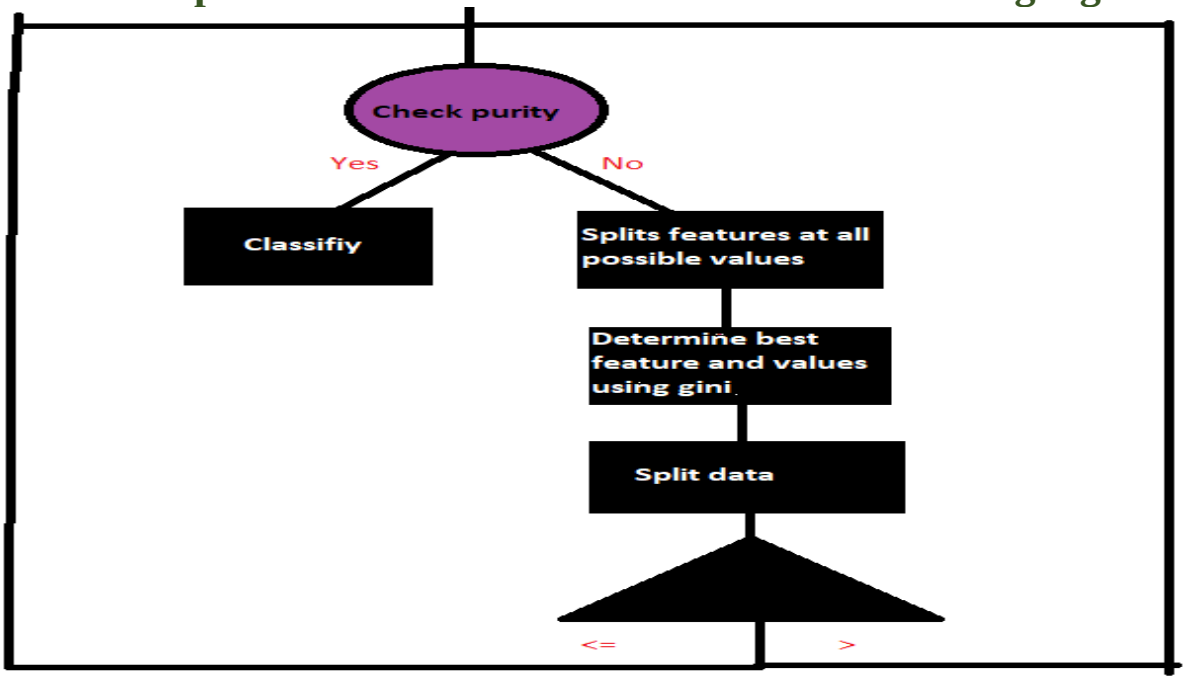
```
data.age.unique()
```

```
array([50, 55, 51, 48, 47, 60, 61, 54, 40, 39, 45, 58, 59, 63, 64, 53, 49,
       57, 56, 46, 43, 62, 52, 42, 44, 41, 29, 30])
```

Final Data

	age	gender	systolic_bp	diastolic_bp	cholesterol	gluc	smoke	alco	active	bmi	cardio
0	2	0	0	1	1	1	0	0	1	1.0	0
1	3	1	3	1	3	1	0	0	1	2.0	1
2	3	1	2	0	3	1	0	0	0	1.0	1

This is simple Chart That I followed when I was building algorithm



Tree Algorithm (Fitting Part)

1- **Check on Three Criteria To know when Stop**

- ***Check Purity, if Current node contain only one kind from classes or sample***

```
def check_purity(data):
    labels = data[:,-1]
    unique_classes = np.unique(labels)

    if len(unique_classes) == 1:
        return True
    else:
        return False
```

- ***Check max depth and Number of samples or data that classify that must be at least 10***

```
def decision_tree_algorithm(data,counter=0,min_sample=10,max_depth=6):
    #data Preperation
    if counter==0:
        global COLUMN_HEADERS
        COLUMN_HEADERS=data.columns
        data=data.values


    else:

        data=data
    # base Algorithm ==> recursive function
    if (check_purity(data)) or (len(data)< min_sample) or(counter==max_depth) :
        classification=classify_data(data)
        return classification
```

- ➢ **If condition of any one True, Algorithm will stop and classify**

```
[1108]
def classify_data(dataset):
    labels = dataset[:,-1]
    unique_classes, count_unique_classes = np.unique(labels, return_counts=True)
    index = count_unique_classes.argmax()
    classification = unique_classes[index]
    return classification
```

```
[1109]
classify_data(train_set[train_set.age<30].values)
```

```
1.0
```

- ➢ **This is Classify function that can classify based on majority class**

2- After Checking on three criteria if condition false, it will follow scenario that I mentioned in picture above

- **Get all possible values for all features of data that can split at**

```
[1161]
def get_potential_split(data):
    potential_splits = {}
    n_cols = data.shape[1]  # Number of columns
    for i_col in range(n_cols - 1): # Disregarding the last label column
        potential_splits[i_col] = []
        values = data[:,i_col]
        unique_values = np.unique(values)   # All possible values
        for index in range(len(unique_values)):
            if index !=0 :
                current_value=unique_values[index]
                previous_value=unique_values[index-1]
                potential_splits[i_col].append((current_value+previous_value)/2)
    return potential_splits
```

```
[1162]
get_potential_split(train_set.values)
```

```
{0: [0.5, 1.5, 2.5, 3.5, 4.5],
 1: [0.5],
 2: [0.5, 1.5, 2.5, 3.5],
 3: [0.5, 1.5, 2.5],
 4: [1.5, 2.5],
 5: [1.5, 2.5],
```

- **This is function that can get all possible splits for all features and its output is dictionary where key is column number and value is list with all possible splits for column**

  - **After that calculate over all gini impurity at each value for each feature and based on the lowest gini, algorithm can know the best value at best feature that can split at (Node)**

```
[117]
def find_best_split(data, potential_splits):
    global best_split_column, best_split_value

    min_overall_impurity = float('inf') # Store the largest overall impurity value
    for coulmn_index in potential_splits:
        for value in potential_splits[coulmn_index]:
            left,right = split_data(data,coulmn_index, value)
            overall_impurity = calculate_overall_gini(left, right)

            if overall_impurity <= min_overall_impurity:    # Find new minimised impurity
                min_overall_impurity = overall_impurity     # Replace the minimum impurity
                best_split_column = coulmn_index
                best_split_value = value
    return best_split_column, best_split_value
```

```
[118]
splits=get_potential_split(train_set.values)

find_best_split(train_set.values,splits)
```

```
(2, 1.5)
```

- **This function to get best value and best feature to split based on gini impurity equation where it split at each value and calculate gini then compare if gini at current value less than previous value, it replaced it and repeat that until get best value at best feature**

Cost Function That is minimized in classification

Overall Gini

$$J = \frac{m_{left}}{m} G_{left} + \frac{m_{right}}{m} G_{right}$$

```
[115]
def calculate_overall_gini(left, right):
    total_num = len(left) + len(right)
    prob_left = len(left) / total_num
    prob_right = len(right) / total_num

    overall_gini = prob_left * calculate_gini(left) + prob_right * calculate_gini(right)

    return overall_gini
```

- **This overall gini Function**

Lowest Overall gini

Gini impurity

$$G = 1 - \sum_{k=1}^{n} p_k^2$$

[1114]

```
def calculate_gini(data):
    labels = data[:,-1]
    _, counts = np.unique(labels, return_counts=True)

    probs = counts / counts.sum()
    gini = 1 - sum(np.square(probs))

    return gini
```

➢ **This Gini value**

- **After getting the best feature and best value ,splitting Node to left and right and doing recursion at each part untill stop at certain layer depend on criteria**

```
def split_data(data,split_column, split_value):
    split_column_values = data[:, split_column]

    left = data[split_column_values <= split_value]
    right = data[split_column_values >  split_value]

    return left, right
```

3]

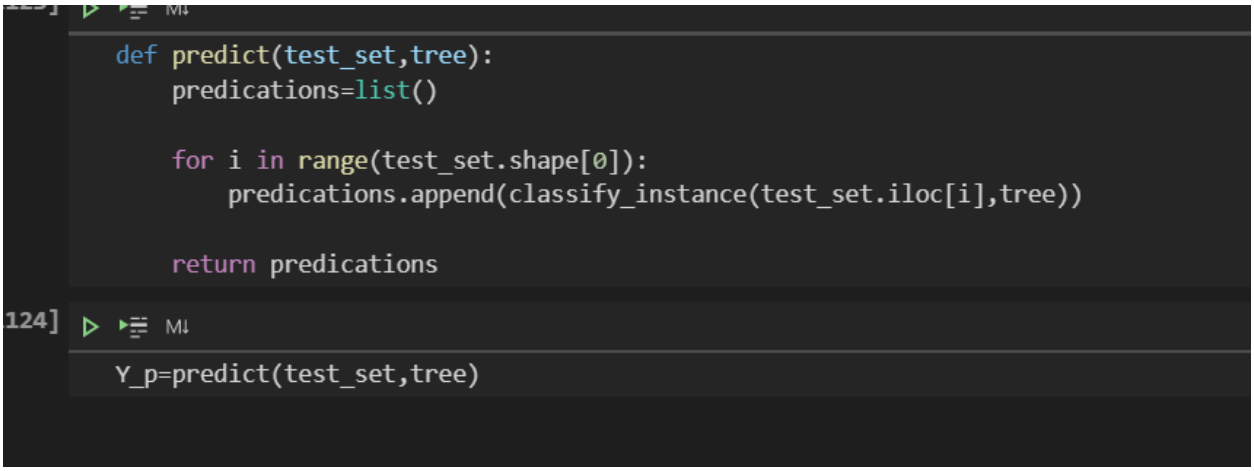
```
left,right=split_data(train_set.values,3,80)
```

➢ **This function to split Node based on best value that calculated at previous step**

3- **Output of Decision Tree Algorithm is Tree or model**

- **Tree is dictionary where key is question that consist of best feature name and operator <= and best split value for this feature**

Prediction Part



```
def predict(test_set,tree):
    predications=list()

    for i in range(test_set.shape[0]):
        predications.append(classify_instance(test_set.iloc[i],tree))

    return predications
```

```
[124]
Y_p=predict(test_set,tree)
```

1- **Predict function loop on classify_instace which its input is tree or model that I built and only one instance and can classify it**

2- **Classify_instance function**

```python
def classify_instance(instance,tree):
        question=list(tree.keys())[0]
        feature_name,comparison_operator,value = question.split()

        #ask question
        if instance[feature_name] <= float(value):
            answer=tree[question][0]
        else:
            answer=tree[question][1]

        #base case
        if not isinstance(answer,dict):
            return answer
        else:
            residual_tree = answer
            return classify_instance(instance,residual_tree)
```

Based on tree key (question that I formed before when I build tree and consist of feature name and operator and value) I check if value of feature for instance less than best value that I split at, to determine where can I go whether in left tree or right

Evaluation Performance

Evaluate Performance

[1125]

```
def accuracy_metric(actual, predicted):
    correct = 0
    for i in range(len(actual)):
        if actual[i] == predicted[i]:
            correct += 1
    return correct / float(len(actual)) * 100.0
```

Accuracy_metric calculate score of model , by see how many instances can classify correctly

Result of My Model

```
random.seed(0)
train_df,test_df=train_test_split(data,test_size=0.1)
Tree=decision_tree_algorithm(train_df,min_sample=10)
y_predict=predict(test_set,tree)
actual=test_df.iloc[:,-1]
accuracy_metric(actual.values,y_predict)
```

```
72.83572958091159
```

It is similar to score of model of sicit_learn

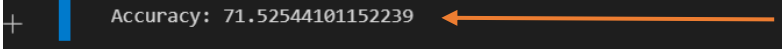
Use Sicit_Learn Model To compare score

[1301]
```
col= data.shape[1]
X= data.iloc[:,:col-1]
Y=data.iloc[:,col-1:col]
```

[1302]
```
X_train, X_test, y_train, y_test = train_test_split(X, Y, test_size=0.3, random_state=1)
```

[1303]
```
Model = DecisionTreeClassifier()
Model.fit(X_train,y_train)
y_pred = Model.predict(X_test)
print("Accuracy:",metrics.accuracy_score(y_test, y_pred)*100)
```

```
Accuracy: 71.52544101152239
```



Bagging Ensemble Learning

Bagging Ensemble Learning.

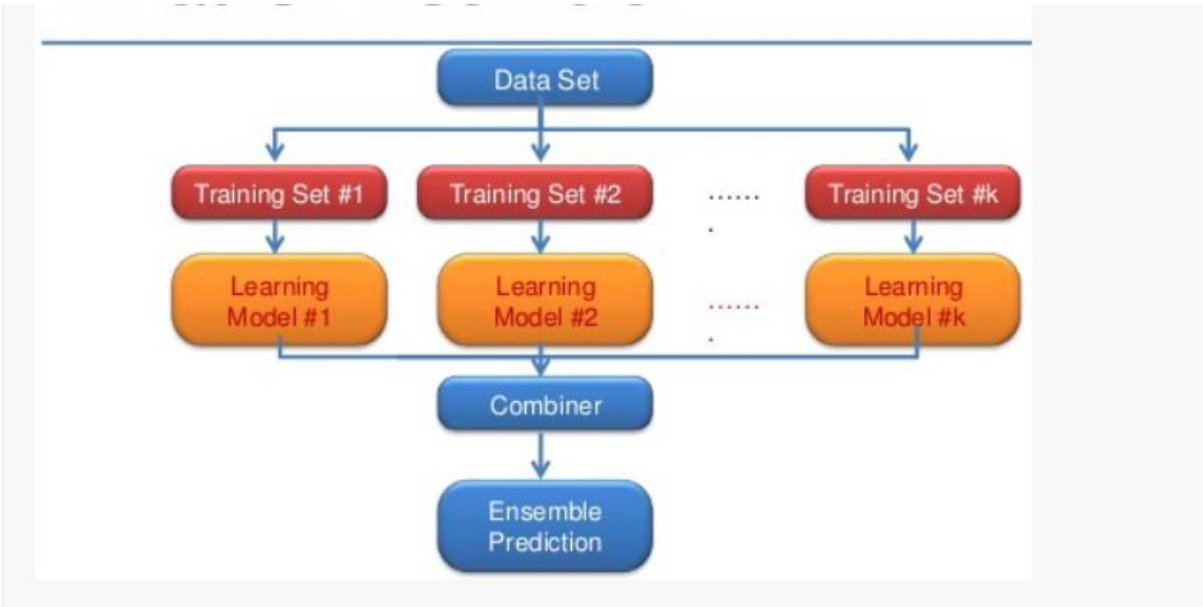
[1304]
```
def Bagged_fitting(data, num_of_bagged):
        models=[]

        for i in range(num_of_bagged):
            sample=data.sample(n=len(data))
            model=decision_tree_algorithm(sample)
            models.append(model)
        return models
```

[1305]

1- IN Bagged_fitting Function, input is data and number of bagged or models
2- I used My Decision Tree Algorithm
3- I sample or select from original data randam data with size L= len(data)
4- Output of Bagged_fitting Function is trees or models that I build

Bagging Prediction

```
def Bagged_prdiction(test_set,models,num_of_bagged):
        pred=np.zeros(len(test_set))
        for model in models:
            pred+=predict(test_set,model)
        return np.round(pred/num_of_bagged)
```



Based on my understanding for this technique, I Used all models that I built and produced Y_prediction for

each model and added all Y_prediction list OF all models to make one prediction

Bagging Result