Bootstrap assignment

There will be some functions that start with the word "grader" ex: grader_sampples(), grader_30().. etc, you should not change those function definition.

Every Grader function has to return True.

Importing packages

In [1]:
```
import numpy as np # importing numpy for numerical computation
from sklearn.datasets import load_boston # here we are using sklearn's boston dataset
from sklearn.metrics import mean_squared_error # importing mean_squared_error metric
```

In [2]:
```
boston = load_boston()
x=boston.data #independent variables
y=boston.target #target variable
```

In [3]:
```
# Lets understand the shape and type of dataset:
print(type(x))
print(type(y))
print(x.shape)
print(y.shape)
```

```
<class 'numpy.ndarray'>
<class 'numpy.ndarray'>
(506, 13)
(506,)
```

Task 1

Step - 1

- **Creating samples**

  Randomly create 30 samples from the whole boston data points

- Creating each sample: Consider any random 303(60% of 506) data points from whole data set and then replicate any 203 points from the sampled points

  For better understanding of this procedure lets check this examples, assume we have 10 data points [1,2,3,4,5,6,7,8,9,10], first we take 6 data points randomly , consider we have selected [4, 5, 7, 8, 9, 3] now we will replicate 4 points from [4, 5, 7, 8, 9, 3], consder they are [5, 8, 3,7] so our final sample will be [4, 5, 7, 8, 9, 3, 5, 8, 3,7]

- Create 30 samples
  - Note that as a part of the Bagging when you are taking the random samples make sure each of the sample will have different set of columns
    Ex: Assume we have 10 columns[1 ,2 ,3 ,4 ,5 ,6 ,7 ,8 ,9 ,10] for the first sample we will select [3, 4, 5, 9, 1, 2] and for the second sample [7, 9, 1, 4, 5, 6, 2] and so on... Make sure each sample will have atleast 3 feautres/columns/attributes

Step - 2

Building High Variance Models on each of the sample and finding train MSE value

- Build a regression trees on each of 30 samples.
- Computed the predicted values of each data point(506 data points) in your corpus.
- Predicted house price of i^{th} data point $y^i_{pred} = \frac{1}{30}\sum_{k=1}^{30}(\text{predicted value of } x^i \text{ with } k^{th} \text{ model})$
- Now calculate the $MSE = \frac{1}{506}\sum_{i=1}^{506}(y^i - y^i_{pred})^2$

Step - 3

- Calculating the OOB score

- Predicted house price of i^{th} data point
  $y^i_{pred} = \frac{1}{k}\sum_{\text{k= model which was buit on samples not included } x^i}(\text{predicted value of } x^i \text{ with } k^{th} \text{ model})$.
- Now calculate the $OOBScore = \frac{1}{506}\sum_{i=1}^{506}(y^i - y^i_{pred})^2$.

Task 2

- **Computing CI of OOB Score and Train MSE**
    - Repeat Task 1 for 35 times, and for each iteration store the Train MSE and OOB score
    - After this we will have 35 Train MSE values and 35 OOB scores
    - using these 35 values (assume like a sample) find the confidence intravels of MSE and OOB Score
    - you need to report CI of MSE and CI of OOB Score
    - Note: Refer the Central_Limit_theorem.ipynb to check how to find the confidence intravel

Task 3

- **Given a single query point predict the price of house.**

Consider xq= [0.18,20.0,5.00,0.0,0.421,5.60,72.2,7.95,7.0,30.0,19.1,372.13,18.60] Predict the house price for this point as mentioned in the step 2 of Task 1.

Task - 1

Step - 1

- **Creating samples**

Algorithm

Pesudo Code for generating Sample

```
def generating_samples(input_data, target_data):

    Selecting_rows <--- Getting 303 random row indices from the input_data

    Replcaing_rows <--- Extracting 206 random row indices from the "Selecting_rows"

    Selecting_columns<--- Getting from 3 to 13 random column indices

    sample_data<--- input_data[Selecting_rows[:,None],Selecting_columns]

    target_of_sample_data <--- target_data[Selecting_rows]

    #Replicating Data

    Replicated_sample_data <--- sample_data [Replaceing_rows]

    target_of_Replicated_sample_data<--- target_data[Replaceing_rows]

    # Concatinating data

    final_sample_data <---  perform vertical stack on  sample_data, Replicated_sample_data

    final_target_data<--- perform vertical stack on target_of_sample_data.reshape(-1,1), target_of_Replicated_sample_data.reshape(-1,1)

    return  final_sample_data,  final_target_data,  Selecting_rows,  Selecting_columns
```

- **Write code for generating samples**

In [4]:
```
# Import necessary libraries:
import random; import math ;


def generating_samples(input_data,target_data):
    #In this function, we will write code for generating 30 samples
```

```
    # Please have a look at this link https://docs.scipy.org/doc/numpy-1.16.0/reference/generated/numpy.random.choice

    # ROW SAMPLING:
    # Randomly generate 60% of 506 data points as indices:
    random.seed(12)
    selected_rows = np.random.choice(len(input_data),math.floor(0.6*len(input_data))
                                     ,replace = False)

    # replicate 40% of 506 data points as indices from selected_rows:
    replicated_rows = np.random.choice(selected_rows,math.ceil(0.4*len(input_data))
                                      ,replace = True)

    # Column sampling without replacement:
    selected_columns = np.random.choice(np.arange(0,13),random.randint(3,13),
                                        replace = False)

    # Create the sample data from selected_rows & columns:
    sample_data = input_data[selected_rows[:,None].tolist(),
                             selected_columns.tolist()]

    # create target feature sample for sampled_rows:
    target_sample = target_data[selected_rows[:,None]]



    # create replicated sample data from replicated_rows & selected_cols :
    replicated_sample = input_data[replicated_rows[:,None].tolist(),
                                   selected_columns.tolist()]

    target_replicated = target_data[replicated_rows[:,None]]

    #concatenating data using vstack()
    sampled_input_data = np.vstack((sample_data,replicated_sample))
    sampled_target_data = np.vstack((target_sample.reshape(-1,1)
                                     ,target_replicated.reshape(-1,1)))

    # Convert the sampled_input & sampled_output data into list
    sampled_input_data = sampled_input_data.tolist()
    sampled_target_data = sampled_target_data.tolist()
    selected_rows = selected_rows.tolist()
    selected_columns = selected_columns.tolist()
```

```
    return sampled_input_data,sampled_target_data,selected_rows,selected_columns
    #returned as lists
```

Grader function - 1 </fongt>

In [5]:
```
def grader_samples(a,b,c,d):
    length = (len(a)==506  and len(b)==506)
    sampled = (len(a)-len(set([str(i) for i in a]))==203)
    rows_length = (len(c)==303)
    column_length= (len(d)>=3)
    assert(length and sampled and rows_length and column_length)
    return True

# this garder function is used to assert our samples requirement and
# we check whether the generated samples return true for this function:
a,b,c,d = generating_samples(x,y)

grader_samples(a,b,c,d)
```

Out[5]:
```
True
```

- **Create 30 samples**

Run this code 30 times, so that you will 30 samples, and store them in a lists as shown below:

```
list_input_data=[]
list_output_data=[]
list_selected_row=[]
list_selected_columns=[]

for i in range(0,30):
    a,b,c,d=generating_sample(input_data,target_data)
   list_input_data.append(a)
   list_output_data.append(b)
   list_selected_row.append(c)
   list_selected_columns.append(d)
```

In [6]:

```
# Use generating_samples function to create 30 samples
# store these created samples in a list

list_input_data =[]
list_output_data =[]
list_selected_row= []
list_selected_columns=[]

# create 30 samples:
for i in range(0,30):
    a,b,c,d = generating_samples(x,y)
    list_input_data.append(a)
    list_output_data.append(b)
    list_selected_row.append(c)
    list_selected_columns.append(d)
```

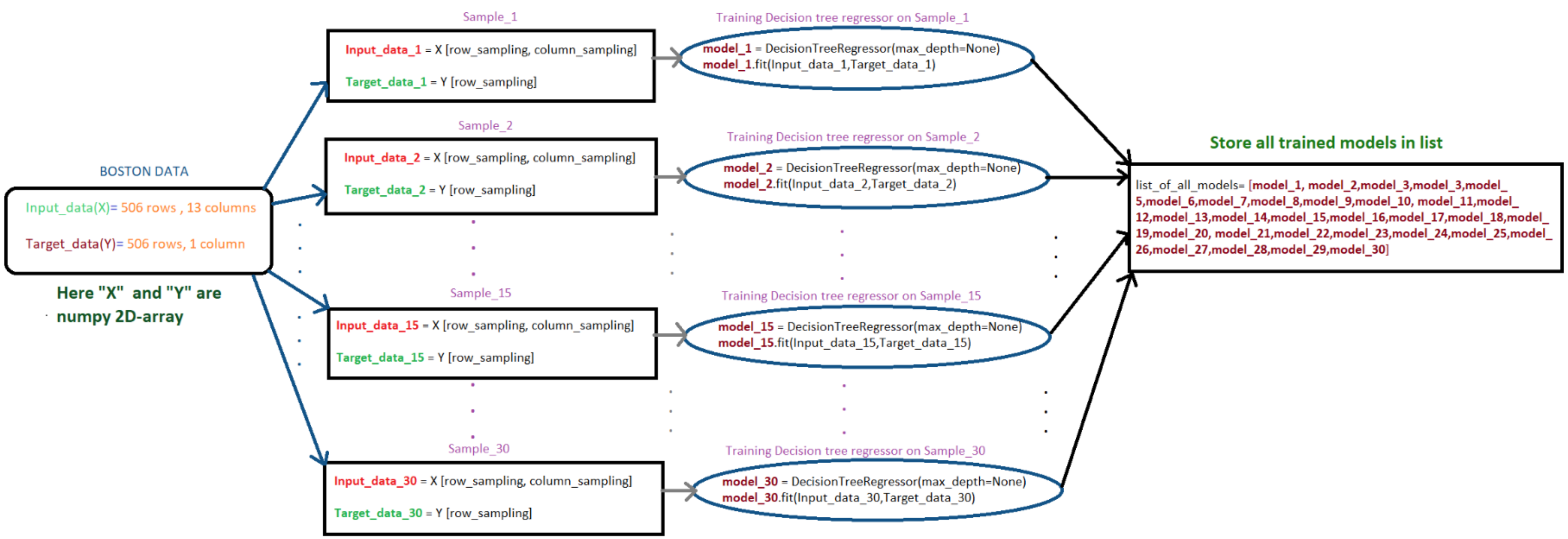
Grader function - 2

In [7]:
```python
def grader_30(a):
    assert(len(a)==30 and len(a[0])==506)
    return True

# # this garder function is used to assert our 30 samples requirement and
# we check whether the generated samples return true for this function:
grader_30(list_input_data)
```

Out[7]: True

Step - 2

Flowchart for building tree

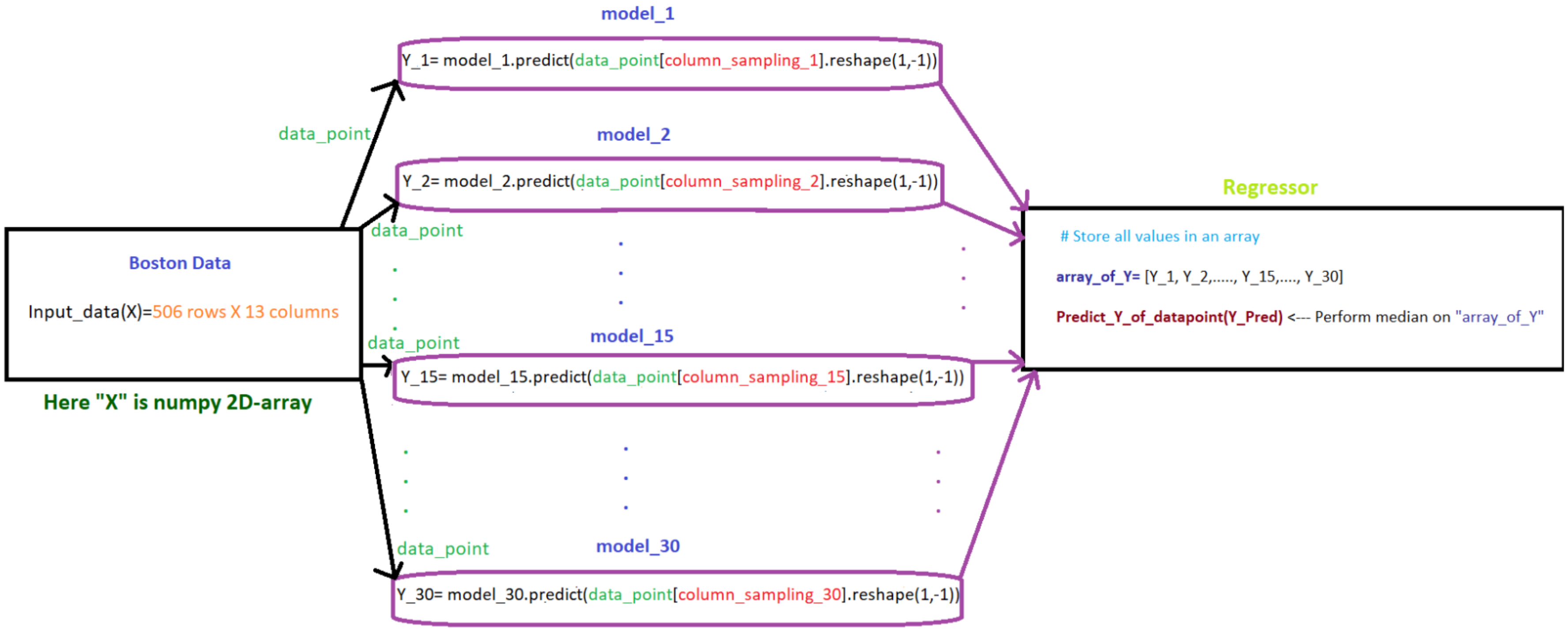


- **Write code for building regression trees**

In [8]:
```
# import necessary library:
from sklearn.tree import DecisionTreeRegressor

# create list to store trained models:
list_of_all_models = []

# Initialize the model with max_depth = None and fit to the 30 samples one by one:
for i in range(0,30):
    model = DecisionTreeRegressor(max_depth = None)
    model.fit(list_input_data[i],list_output_data[i])
    list_of_all_models.append(model)

# print the following:
print("The number of base learners trained : ",len(list_of_all_models))
print("The list of all trained models : \n",list_of_all_models)
```

```
The number of base learners trained :  30
The list of all trained models : 
 [DecisionTreeRegressor(), DecisionTreeRegressor(), DecisionTreeRegressor(), DecisionTreeRegressor(), DecisionTreeReg
ressor(), DecisionTreeRegressor(), DecisionTreeRegressor(), DecisionTreeRegressor(), DecisionTreeRegressor(), Decisio
nTreeRegressor(), DecisionTreeRegressor(), DecisionTreeRegressor(), DecisionTreeRegressor(), DecisionTreeRegressor(),
DecisionTreeRegressor(), DecisionTreeRegressor(), DecisionTreeRegressor(), DecisionTreeRegressor(), DecisionTreeRegre
ssor(), DecisionTreeRegressor(), DecisionTreeRegressor(), DecisionTreeRegressor(), DecisionTreeRegressor(), DecisionT
reeRegressor(), DecisionTreeRegressor(), DecisionTreeRegressor(), DecisionTreeRegressor(), DecisionTreeRegressor(), D
ecisionTreeRegressor(), DecisionTreeRegressor()]
```

Flowchart for calculating MSE



After getting predicted_y for each data point, we can use sklearns mean_squared_error to calculate the MSE between predicted_y and actual_y.

- **Write code for calculating MSE**

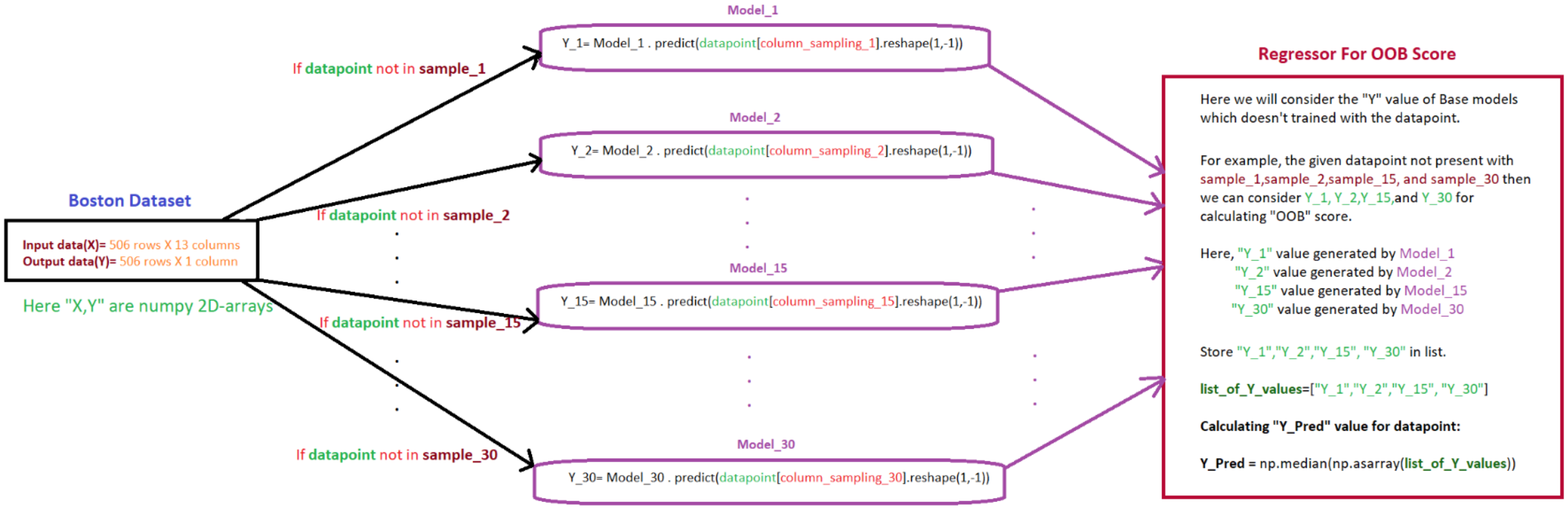
In [9]:
```
# Necessary libraries for the desired task:
from sklearn.metrics import mean_squared_error
from statistics import median
from tqdm import tqdm
from statistics import mean

# code to predict each of the training instances:
array_of_pred_y = [] # to store final aggregated output/result
```

```
for k in tqdm(range(0,len(x))):
    array_of_y = []      # to store each base learners output for a particular data point
    for j in range(0,30):
        y_j = list_of_all_models[j].predict(x[k,list_selected_columns[j]].reshape(1,-1))
        array_of_y.append(y_j) # append output of each base learners

    # compute median as part of aggregation stage:
    y_pred = median(array_of_y)
    array_of_pred_y.append(y_pred) # append final predicted values

print("The number of predicted values :",len(array_of_pred_y))

# create a custom function for computing mse:
def mse(y,ypred):
    sum_diff = 0
    for i in range(0,len(x)):
        diff = (y[i] - ypred[i][0])
        sum_diff += (diff*diff)
    value = (sum_diff)/len(x)
    return value

# call the mse() and return the mse & rmse values:
val = mse(y.tolist(),array_of_pred_y)
print("MSE value :",val)
print("RMSE value :",math.sqrt(val))
```

```
100%|██████████████████████████████████████████████████████████████████████████████| 506/506 [00:02<00:00, 250.45it/s]
The number of predicted values : 506
MSE value : 0.04883170417226433
RMSE value : 0.22097896771472242
```

Step - 3

Flowchart for calculating OOB score



Now calculate the $OOBScore = \frac{1}{506}\sum_{i=1}^{506}(y^i - y^i_{pred})^2$.

- **Write code for calculating OOB score**

In [10]:
```
# necessary library for desired task:
from tqdm import tqdm

# To calculate oob score:
pred_y_oob = []             # store oob final predicted values.

for i in tqdm(range(0,len(x))):
    models_y_pred = []      # store oob scores of respective base learners.

    for k in range(0,len(list_of_all_models)): # iterate through 30 base learners
        if i not in list_selected_row[k]:
```

```
            models_y_pred.append(list_of_all_models[k].
                                 predict(x[i,list_selected_columns[k]].reshape(1,-1)))
        else:
            continue
    # compute final aggregated score & append it in a list
    pred_y_oob.append(median(models_y_pred))


print("The number of predicted oob scores :",len(pred_y_oob))

# create a custom function for computing mse of oob points:
def mse(y,ypred):
    sum_diff = 0
    for i in range(0,len(x)):
        diff = (y[i] - ypred[i][0])
        sum_diff += (diff*diff)
    value = (sum_diff)/len(x)
    return value

# call the mse() and return the mse(oob) & rmse(oob) values:
val = mse(y.tolist(),pred_y_oob)
print("OOB score (MSE) :",val)
print("OOB score (RMSE) :",math.sqrt(val))
```

```
100%|████████████████████████████████████████████████████████████████████████████████| 506/506 [00:00<00:00, 549.46it/
s]
The number of predicted oob scores : 506
OOB score (MSE) : 13.518175852075473
OOB score (RMSE) : 3.6767072023857805
```

Task 2

* **Write code for calculating confidence interval**

In [11]:
```
# import required libraries
from tqdm import tqdm

# define a function to perform task(2)
def task2():
    list_of_all_models = []
```

```python
    # To compute mse scores:
    for i in range(0,30):
        model = DecisionTreeRegressor(max_depth = None)
        model.fit(list_input_data[i],list_output_data[i])
        list_of_all_models.append(model)

    array_of_pred_y = []
    for k in range(0,len(x)):
        array_of_y = []
        for j in range(0,30):
            y_j = list_of_all_models[j].predict(x[k,list_selected_columns[j]].reshape(1,-1))
            array_of_y.append(y_j)

        y_pred = median(array_of_y)
        array_of_pred_y.append(y_pred)

    val_mse = mse(y.tolist(),array_of_pred_y)

    # To calculate oob score:
    pred_y_oob = []
    for i in range(0,len(x)):
        models_y_pred = []
        for k in range(0,len(list_of_all_models)):
            if i not in list_selected_row[k]:
                models_y_pred.append(list_of_all_models[k].
                                 predict(x[i,list_selected_columns[k]].reshape(1,-1)))

        pred_y_oob.append(median(models_y_pred))

    val_oob = mse(y.tolist(),pred_y_oob)

    return [val_mse,val_oob] # return the scores

mylist = []
for i in tqdm(range(0,35)):
    scores = task2()
    mylist.append(scores)

print("The number of iterations :",len(mylist))
print("The list of MSE & OOB Scores :\n",mylist)
```

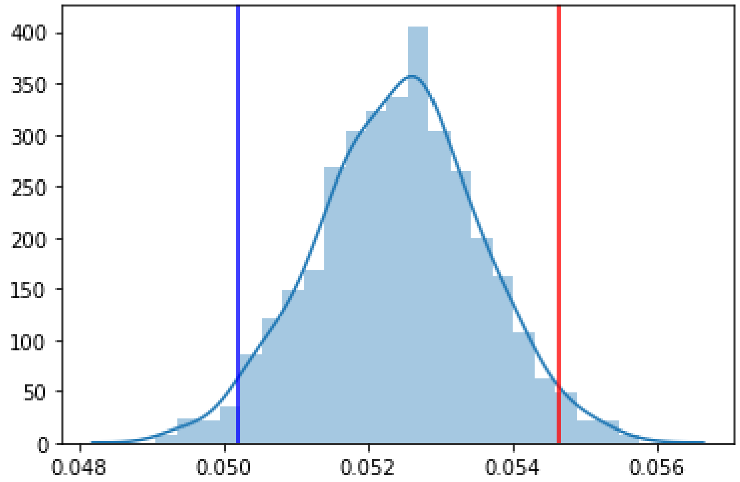
```
100%|████████████████████████████████████████████████████████████████| 35/35 [01:18<00:00,  2.25s/i
t]
The number of iterations : 35
The list of MSE & OOB Scores :
 [[0.055740778748424764, 12.940639137588226], [0.05523715415019764, 12.729159867886999], [0.05251033834002891, 13.209
098705473348], [0.059557752837138, 13.053052291073366], [0.056191168379554665, 13.390863342470627], [0.05174946877481
1534, 12.892452282065094], [0.04520096979664051, 13.302713242141026], [0.05395776518891667, 13.738759908765731], [0.0
5618208892903519, 14.133212267253302], [0.05941331422547792, 12.784001254181817], [0.05740048470860601, 13.4846728788
30688], [0.049382083096250834, 13.404201784309016], [0.05021706333340502, 12.959799791633499], [0.04844609275808662,
13.716175680814128], [0.03919916334901204, 14.123529760759903], [0.046956080345059444, 13.570672858460076], [0.064203
41118694042, 13.331802762471192], [0.04811506509010245, 13.864559100798363], [0.04746289117705897, 13.15322683325009
7], [0.050884411063422605, 12.590888146702255], [0.05054589512962815, 13.302462351599344], [0.046474086493500426, 13.
472652820770715], [0.06133154545077442, 13.644201996579076], [0.038211132565634784, 13.30033045835175], [0.0461193799
40711504, 13.075922959590333], [0.05397992551613788, 12.95391104113345], [0.050862100663224975, 13.486097586952111],
[0.06226877096680426, 13.405168809672892], [0.06303979643891659, 13.767537331652523], [0.045825176012845865, 13.24900
2590913221], [0.06405874097943053, 13.236190609467828], [0.048631422924901226, 13.328348314292908], [0.04275439315334
356, 12.793425838255787], [0.05112897628081387, 13.724346825938683], [0.060281468459154976, 13.178256505959533]]
```

* **Obtain the 95% confidence intervals**

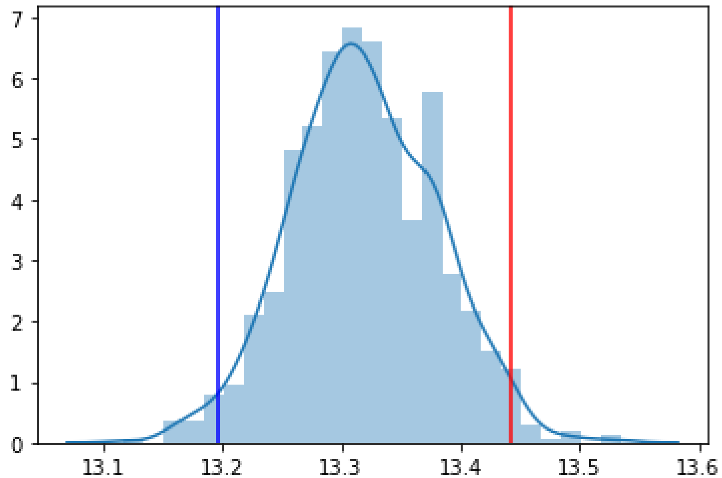
In [16]:
```python
# Import necessary libraries:
from statistics import mean
from statistics import pstdev
import seaborn as sns
import matplotlib.pyplot as plt
import math

# reference: Central_Limit_theorem.ipynb
mse_values = []
oob_scores = []

# unpack mylist to seperate lists of scores:
for scores in mylist:
    mse_values.append(scores[0])
    oob_scores.append(scores[1])
#print(mse_values)
#print(oob_scores)

# As we already know the pop_mean and pop std,
# we will store them for finding 95% conf_interval & interpret them .
pop_mse_mean = mean(mse_values)
pop_oob_mean = mean(oob_scores)
```

```
pop_mse_std = pstdev(mse_values)
pop_oob_std = pstdev(oob_scores)

# define custom function to create sampling distribution of sample means:
def get_means_of_n_samples_with_m_size(data,n,m):
    sample_mean_m_samples_n_ele = []
    for i in range(0,n):
        samples = np.random.choice(data,m,replace = True)
        sample_mean_m_samples_n_ele.append(mean(samples.tolist()))
    return sample_mean_m_samples_n_ele

# Lets create 500 samples with sample size = 35 & store it in a variable:
sampling_set_mse = get_means_of_n_samples_with_m_size(mse_values,1000,35)
sampling_set_oob = get_means_of_n_samples_with_m_size(oob_scores,1000,35)


# define function to find 95% confidence interval & plot the distribution :
def conf_interval(sampling_dist,pop_std,m):
    lowlimit = mean(sampling_dist) - (1.96 * (pop_std/math.sqrt(m)))
    upplimit = mean(sampling_dist) + (1.96 * (pop_std/math.sqrt(m)))
    print("\nThe sampling distribution of sample means :")
    ax = sns.distplot(sampling_dist)
    ax.axvline(lowlimit, color='b')
    ax.axvline(upplimit, color='r')
    plt.show()
    return [lowlimit,upplimit]

# Reporting confidence interval of MSE Scores:
mse_interval = conf_interval(sampling_set_mse,pop_mse_std,35)
print("The true population mean of train_mse is :",pop_mse_mean)
print("The 95% confidence interval of train_MSE : ",mse_interval)

# Reporting confidence interval of OOB Scores:
oob_interval = conf_interval(sampling_set_oob,pop_oob_std,35)
print("The true population mean of oob_score is :",pop_oob_mean)
print("The 95% confidence interval of oob_score : ",oob_interval)
```

```
The sampling distribution of sample means :
```




```
The true population mean of train_mse is : 0.05238629589868552
The 95% confidence interval of train_MSE :  [0.05019447379044782, 0.05463289012317454]

The sampling distribution of sample means :
```

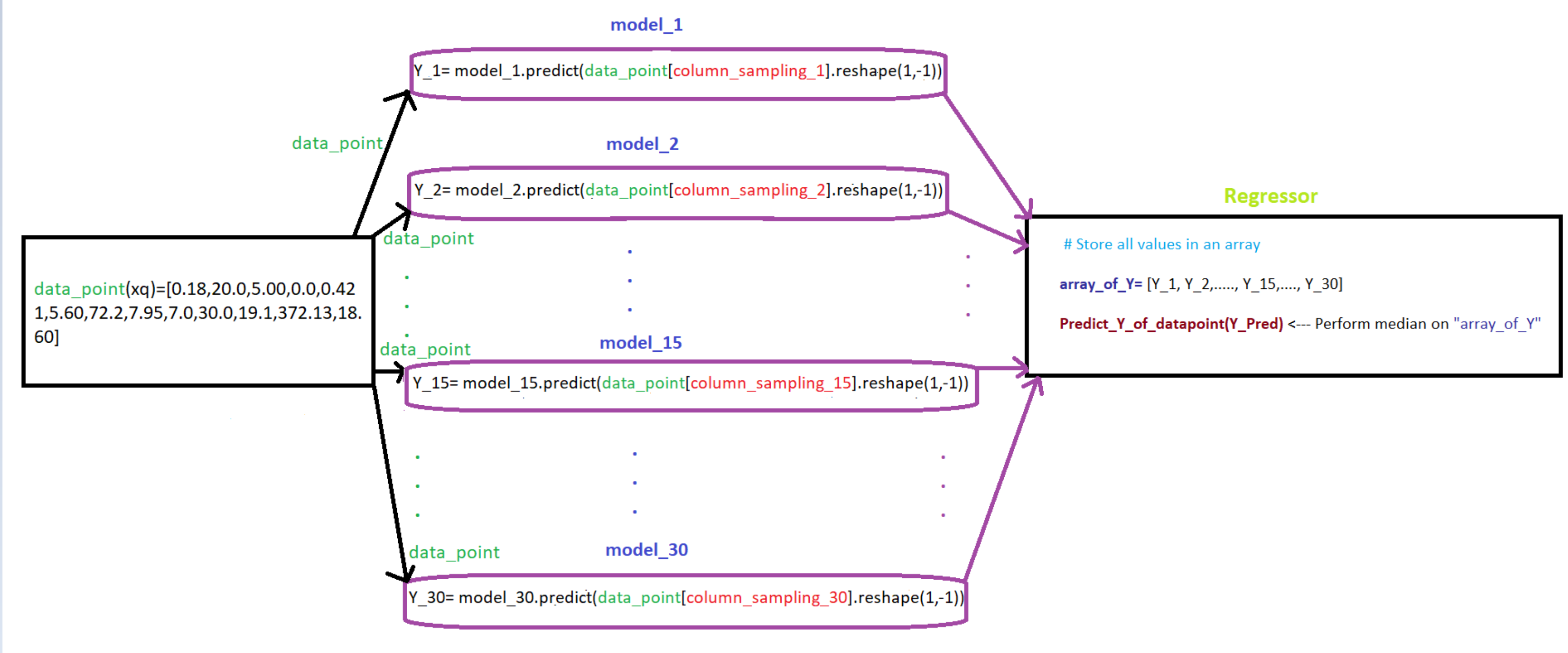



```
The true population mean of oob_score is : 13.322609655373112
The 95% confidence interval of oob_score :  [13.196540281406497, 13.442041433641768]
```

Task 3

Flowchart for Task 3

Hint: We created 30 models by using 30 samples in TASK-1. Here, we need send query point "xq" to 30 models and perform the regression on the output generated by 30 models.



- **Write code for TASK 3**

In [19]:
```
# Import necessary libraries:
from statistics import median

# define a query point xq:
xq = [0.18,20.0,5.00,0.0,0.421,5.60,72.2,7.95,7.0,30.0,19.1,372.13,18.60]
datapoint = np.array(xq)

# lets pass the xq to each of the base learners :
y_array = []
for i,model in enumerate(list_of_all_models):
    y_j = model.predict(datapoint[list_selected_columns[i]].reshape(1,-1))
```

```
    y_array.append(y_j)

# Aggregate the results:
ypred = median(y_array)

print("The predicted house price :",ypred)
```

```
The predicted house price : [18.85]
```

Write observations for task 1, task 2, task 3 indetail

OBSERVATIONS:

1) From task (1) we can observe that

Train_mse value = 0.0488 and Oob score_mse value = 13.518 where, we can clearly see high value of oob_score which indicates that oob sample does not serve as a good validation set for checking the efficiency of random forest model. Here, since the oob_samples are evaluated with very few base learners -- > the final aggregation result is not robust in nature & thus validation on full ensemble of DT's is better than oob_samples.

2) From task (2) we can observe that

The 95% confidence interval for both mse_score & oob_score includes the true population means and also the sampling distribution of sample means follows normal distribution as part of central limit theorem.

3) From task (3) we can observe that

The given query point (xq) being tested against each base learner and we finally get the aggregated output which serves as the predicted house price.