ENSEMBLES

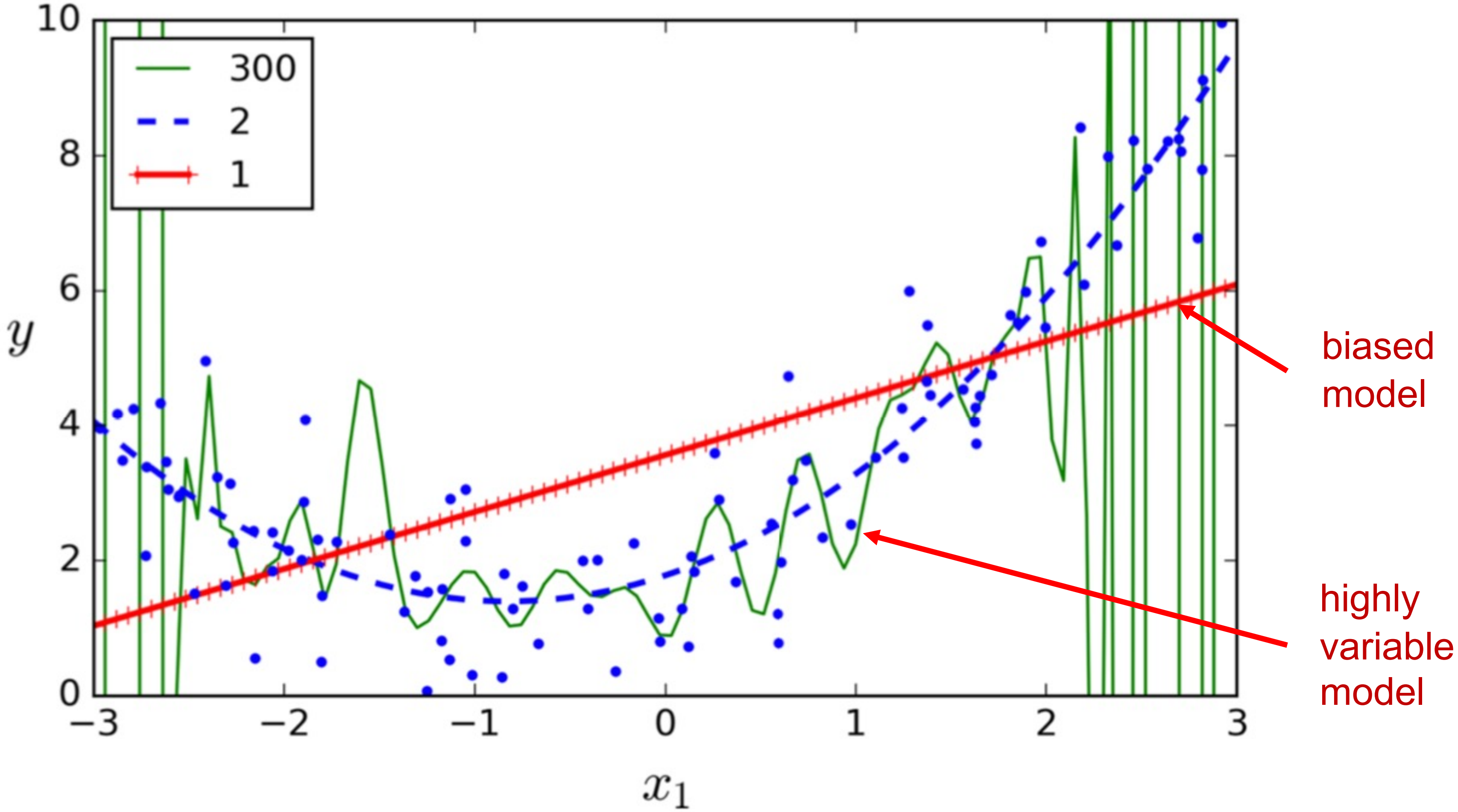
Outline

- Definition of Ensembles
- Bias-Variance Tradeoff
- Bootstrap samples
- Ensembles
- Bagging
- Random Forest
- Gradient Boosting

Ensembles

- Methods combining multiple machine learning models to create low-bias, low-variance, prediction models
- What are low-bias and low-variance models?

Bias – Variance Tradeoff



Bias – Variance Tradeoff

The model error (MSE, error rate) is the sum of 3 parts

- Bias
- Variance
- Random error

Bias – Variance Tradeoff

The model error (MSE, error rate) is the sum of 3 parts

- Bias

Model is unable to follow the structural variation underlying the data. (i.e., a linear model used with nonlinear data).

Models with high-bias are likely to underfit the data.

Bias – Variance Tradeoff

The model error (MSE, error rate) is the sum of 3 parts

- Bias

Model is unable to follow the structural variation underlying the data. (i.e., a linear model used with nonlinear data).
Models with high-bias are likely to underfit the data.

- Variance

Model is highly sensitive, trying to capture random variations in the data. (i.e., a high-degree polynomial model).
Models with high-variance are likely to overfit the data.

Bias – Variance Tradeoff

The model error (MSE, error rate) is the sum of 3 parts

- Random error

  Data portion that cannot be predicted

We look for models that do not underfit or overfit the data

These are low-bias, low-variance models

Model Complexity

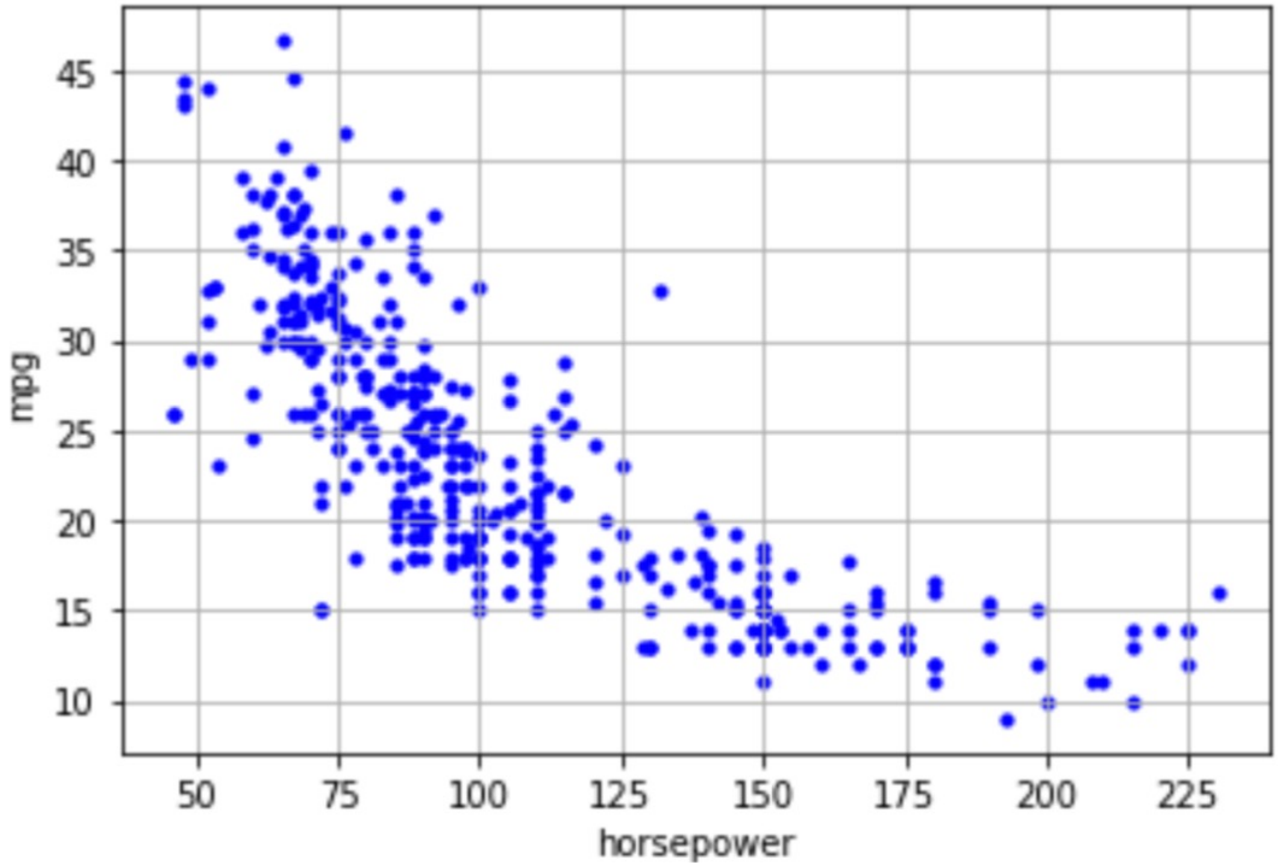
- Model Complexity is given by the number of predictor columns in the model.
- Adding new predictors, transformed predictors, interactions, and polynomial terms increases the model complexity

Model Complexity

- Increasing model complexity usually increases the model's variance and reduces its bias.
- Decreasing model complexity usually increases the model's bias and reduces its variance.
- This relation between the error portions of a model is called the Bias – Variance Tradeoff

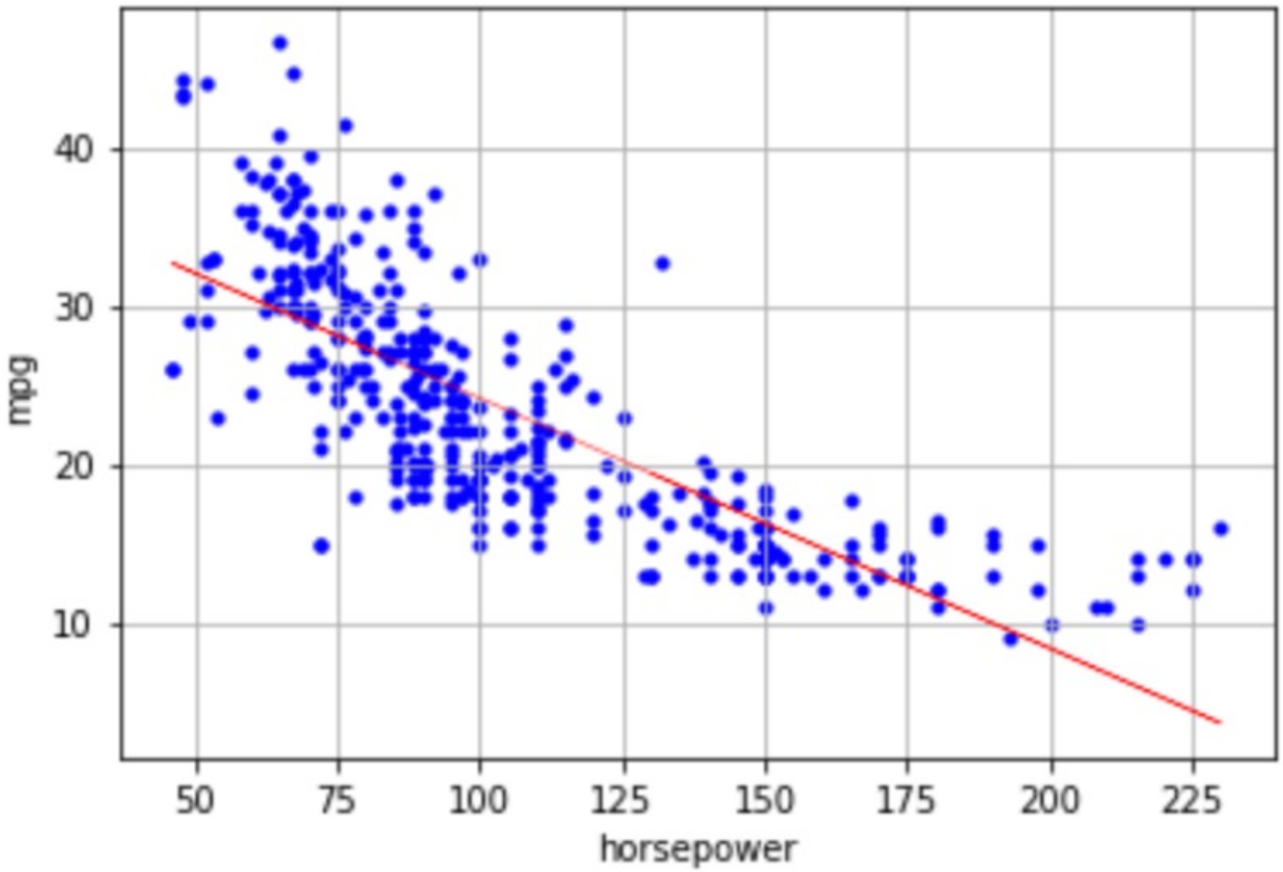
Bias – Variance Tradeoff

Non-linear relation



Bias – Variance Tradeoff

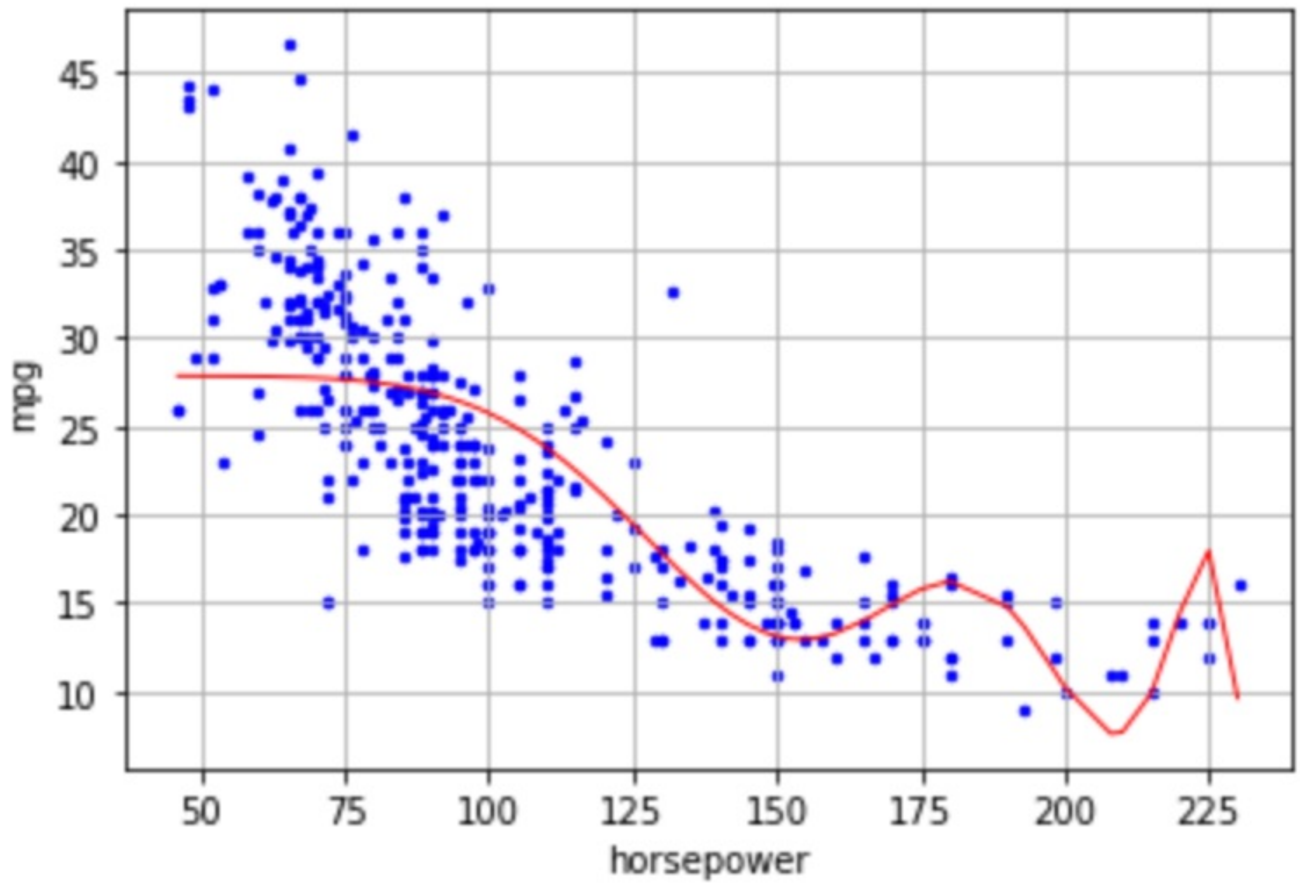
Model with High-Bias (does not follow underlying pattern)



underfits the data

Bias – Variance Tradeoff

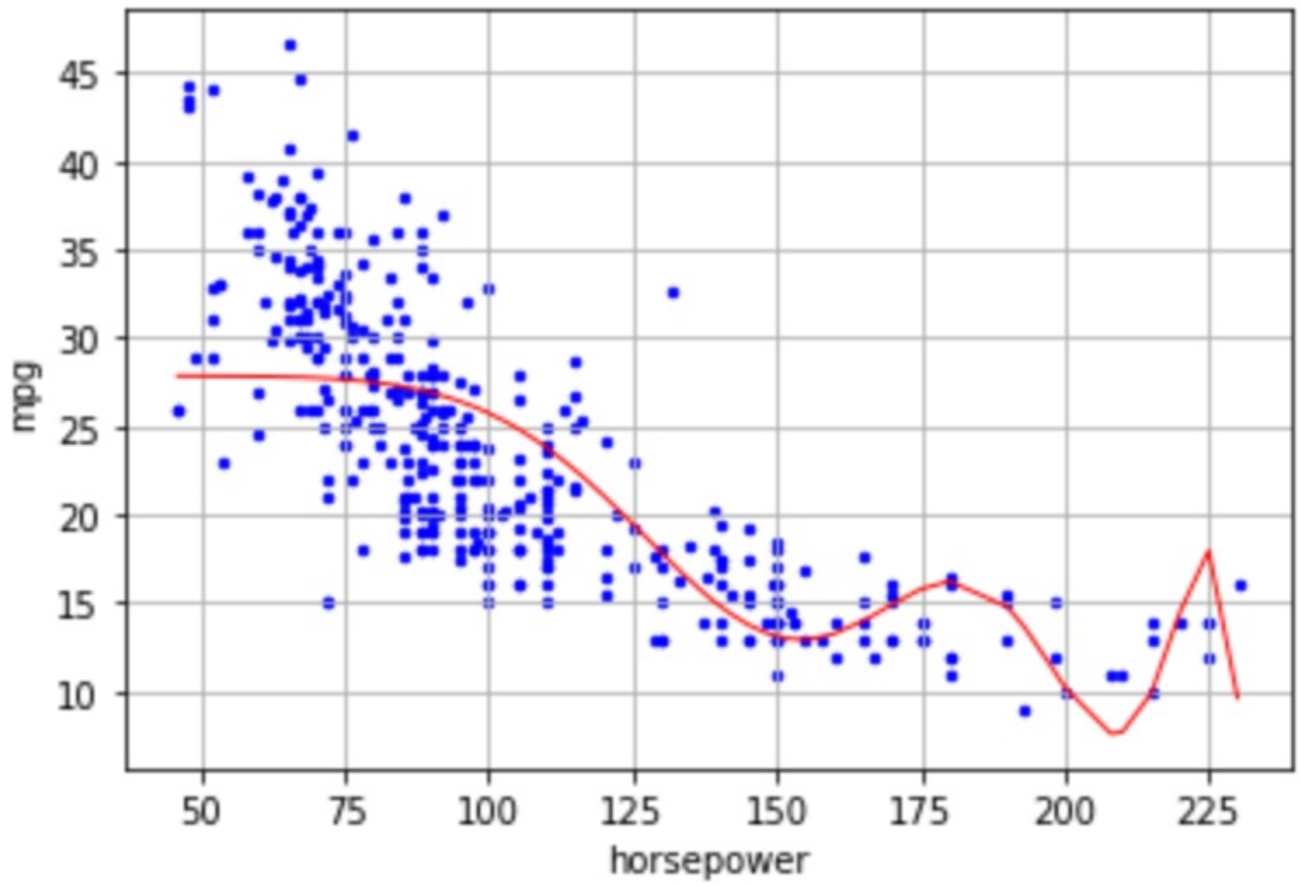
Model with High-Variance (high-degree polynomial)



overfits the data

Bias – Variance Tradeoff

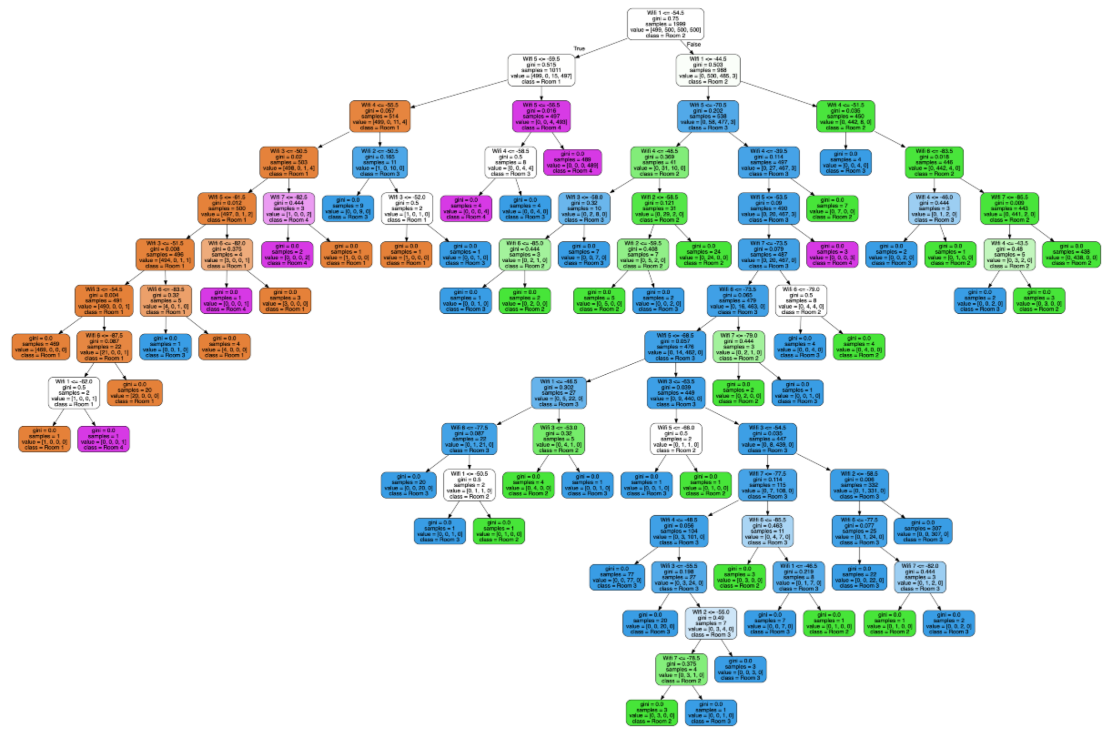
Model with High-Variance (high-degree polynomial)



the model follows both the underlying pattern and the random errors in the data

Tree complexity

- A decision Tree with a large depth is a high variance model
- To avoid it, we may define a maximum depth
- But then predictions may become less accurate



Balancing Tree Complexity

- Ensembles are combination of models (trees)
- They tend to be low-bias, low-variance models

ENSEMBLES

Ensembles

- Methods combining multiple ML models to create low-bias, low-variance, models
- They combine multiple models to create new more accurate models
- Types of ensembles of trees
    - Bagged trees
    - Random Forest
    - Gradient boosting trees

Hyperparameters

Random Forest

- max_features
- n_estimators
- max_depth

Gradient Boosting

- learning_rate
- max_features
- n_estimators
- max_depth

BAGGING

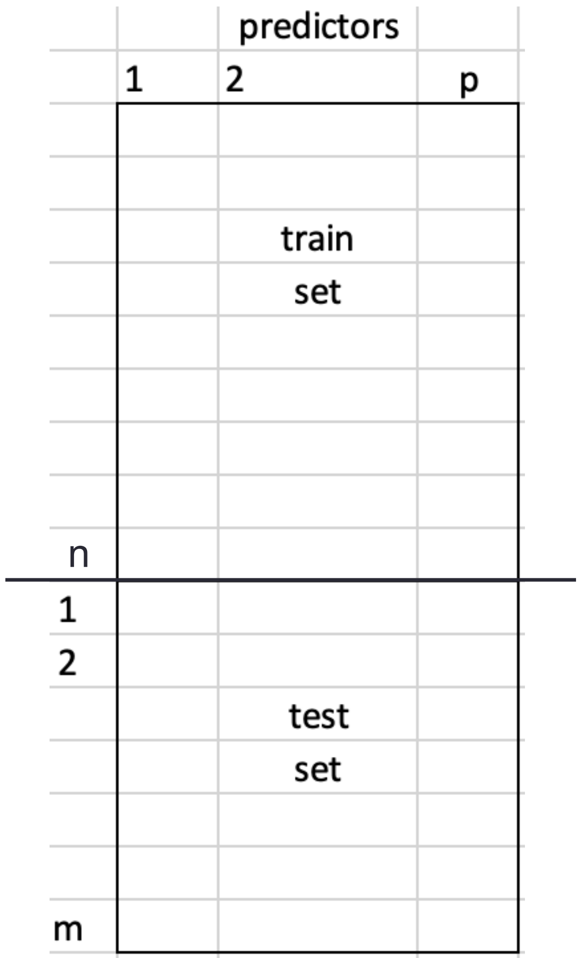
Bootstrap samples (from dataframes)

- A bootstrap sample is a sample *with* replacement
- May include same row many times
- Bootstrap samples are usually of the same size

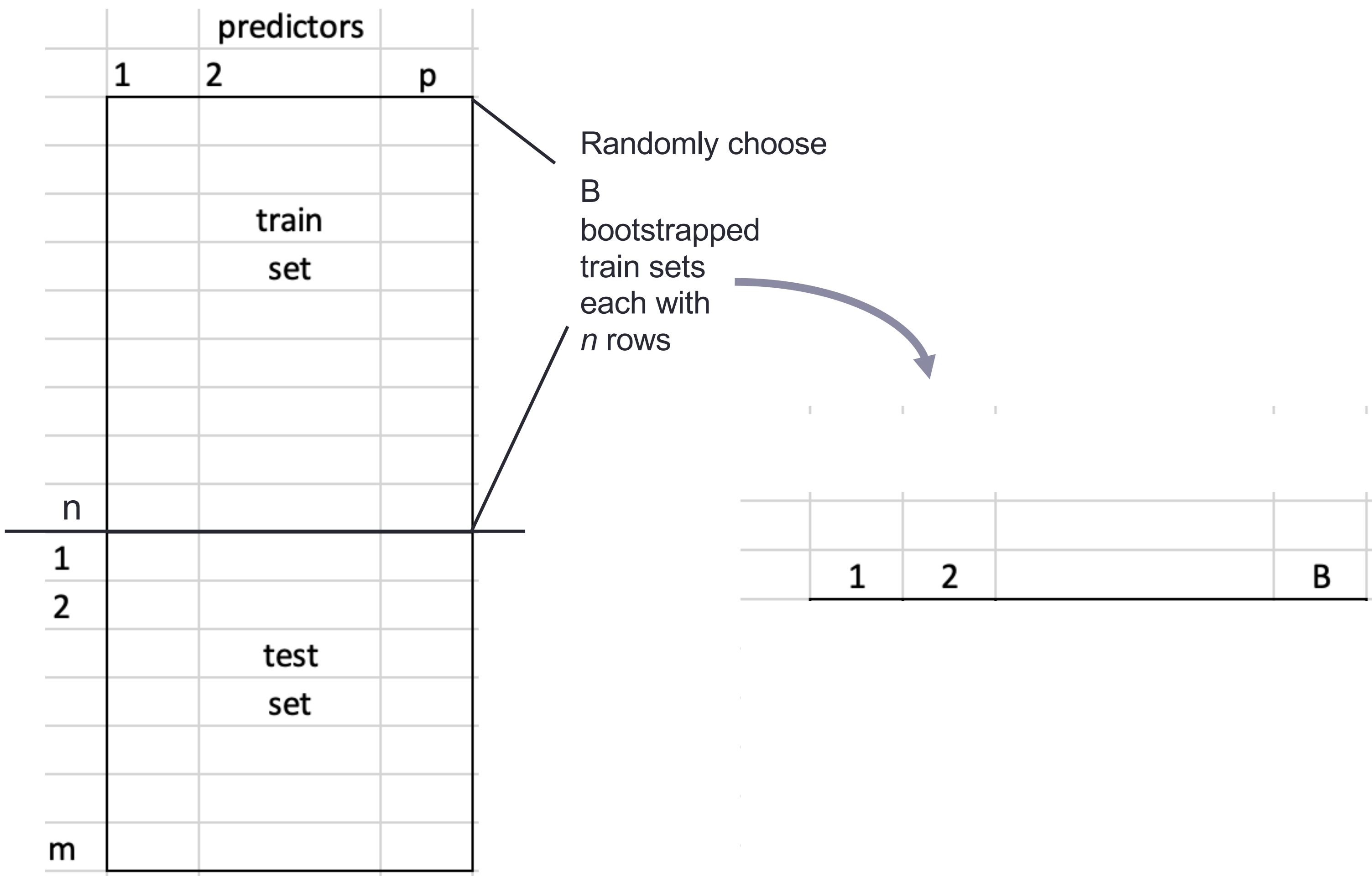
Bagging

- Individual trees suffer from high variance
- That is, if a dataset is randomly split into 2 sets and a tree is fit to each half, the predictions may be *very* different
- On the other hand, a low-variance model would yield predictions that are not much different
- Averaging trees predictions help avoid high-variance models

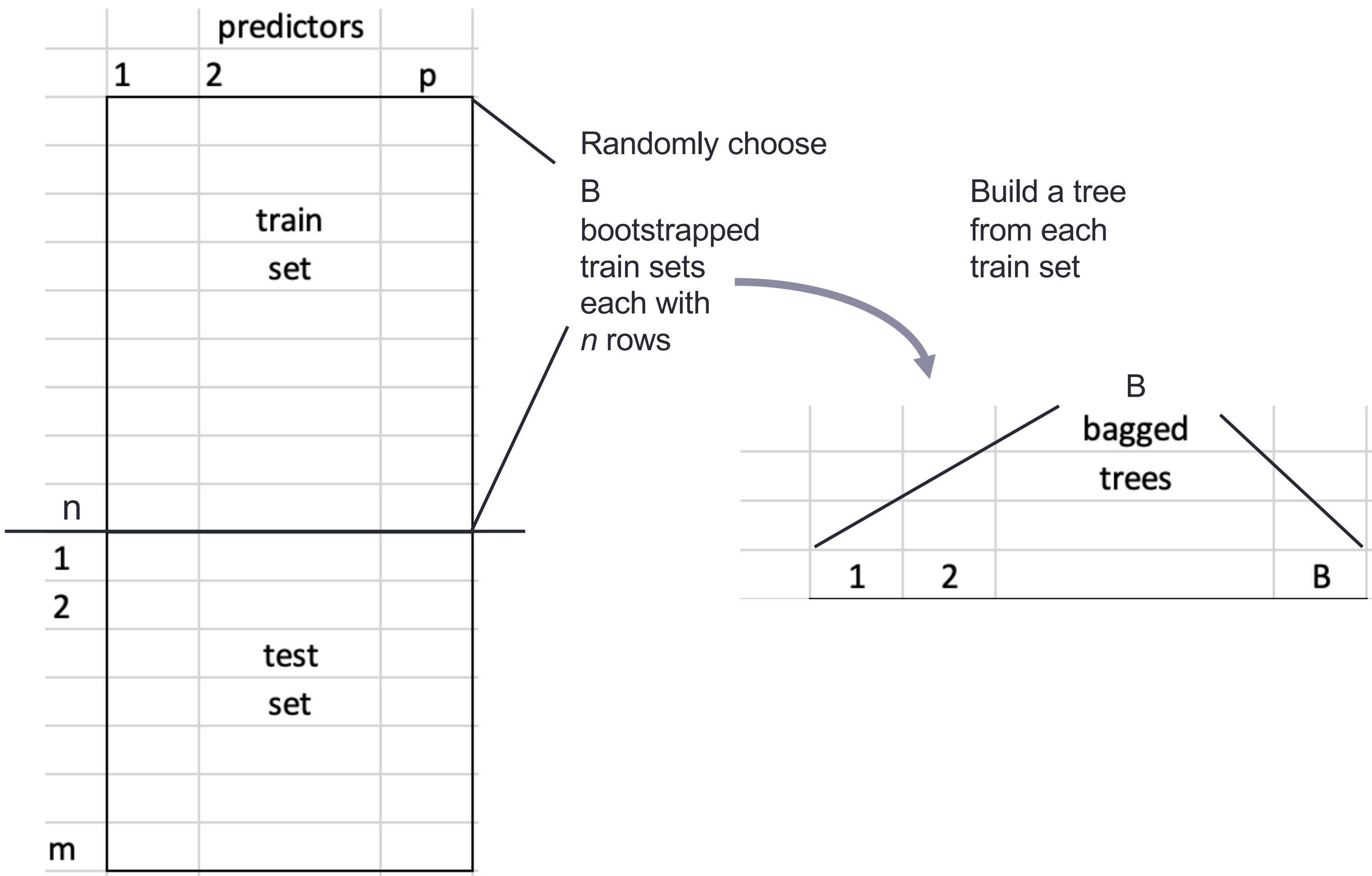
Bagging for **Regression Trees**



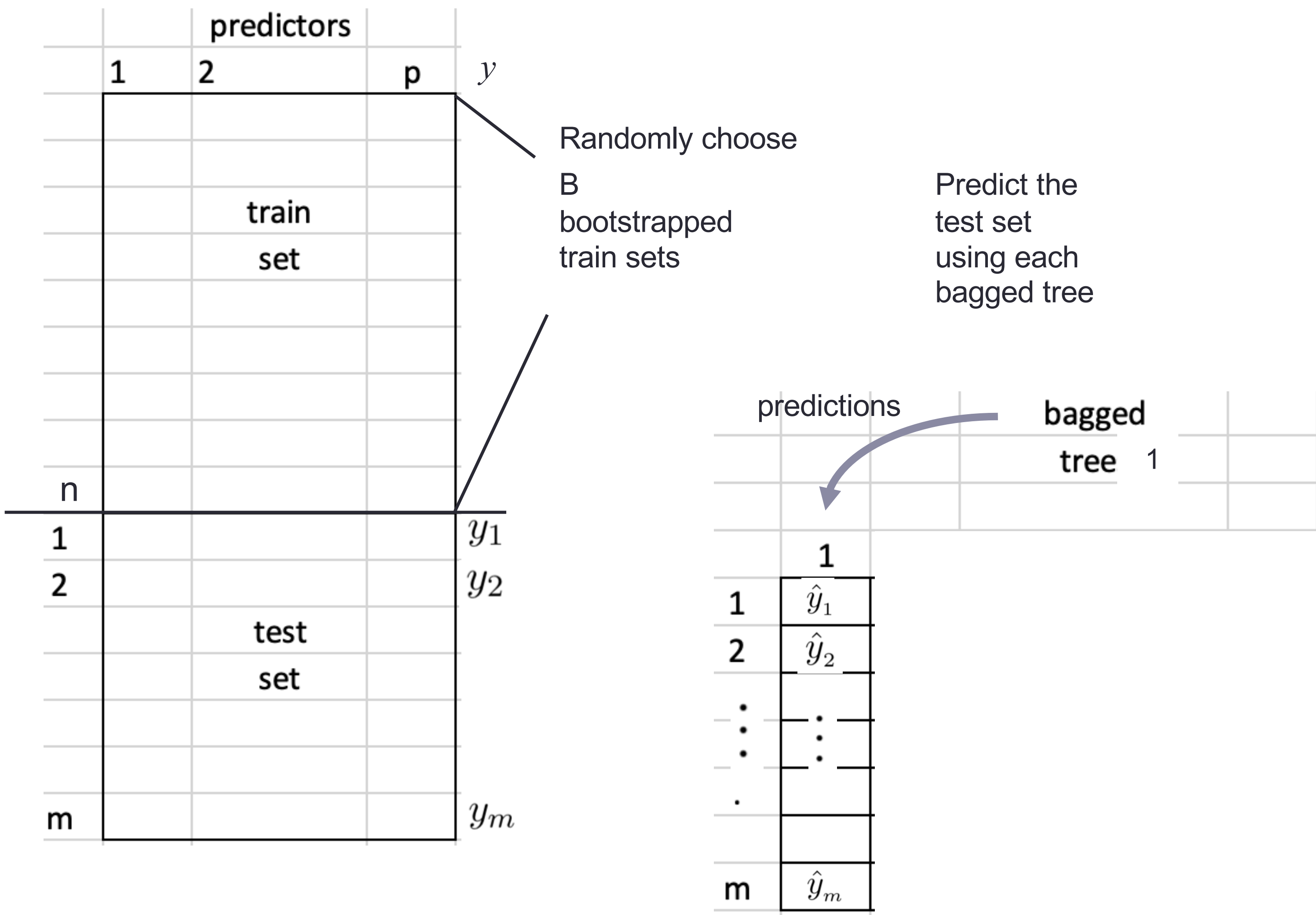
Bagging for Regression Trees



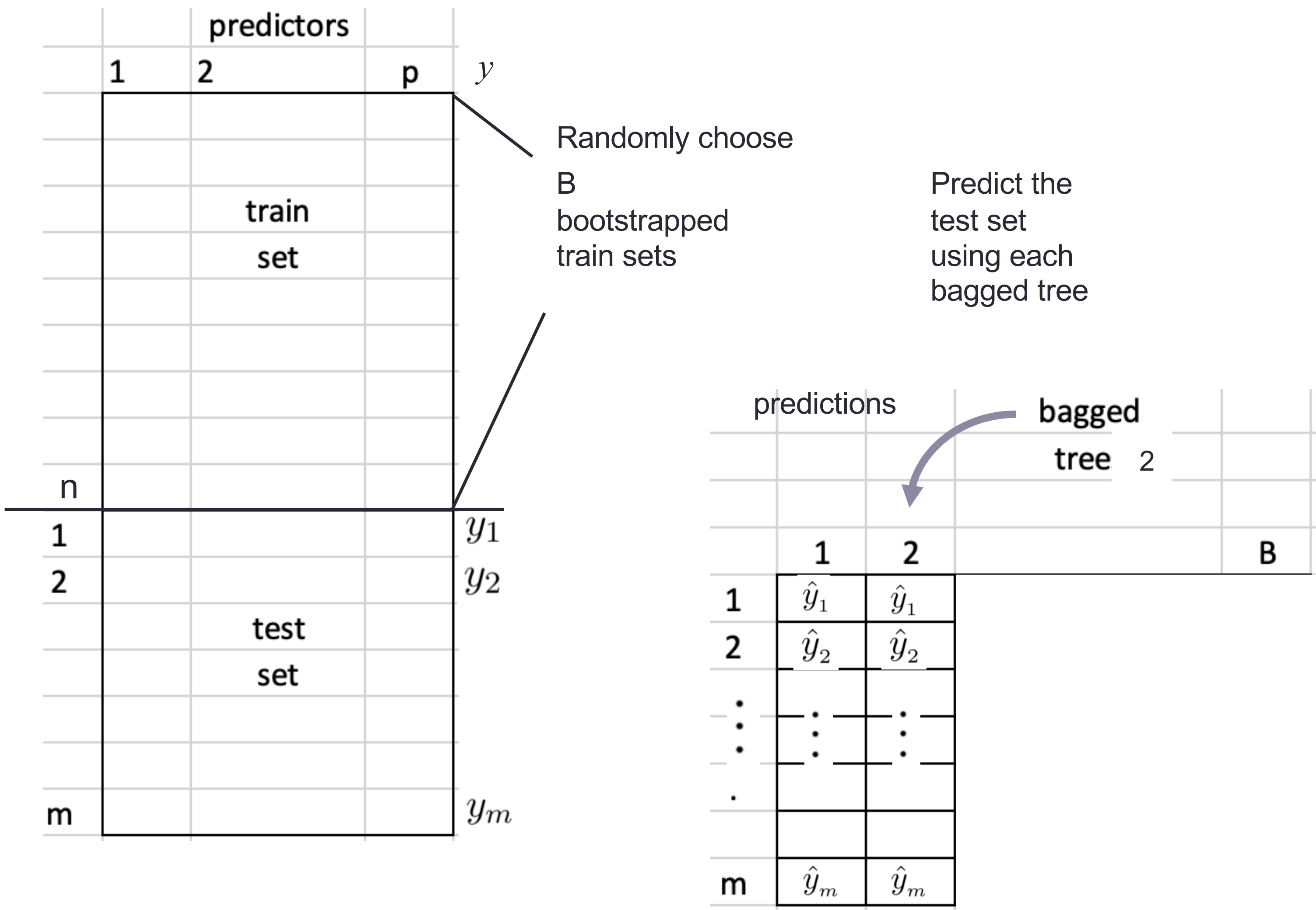
Bagging for Regression Trees



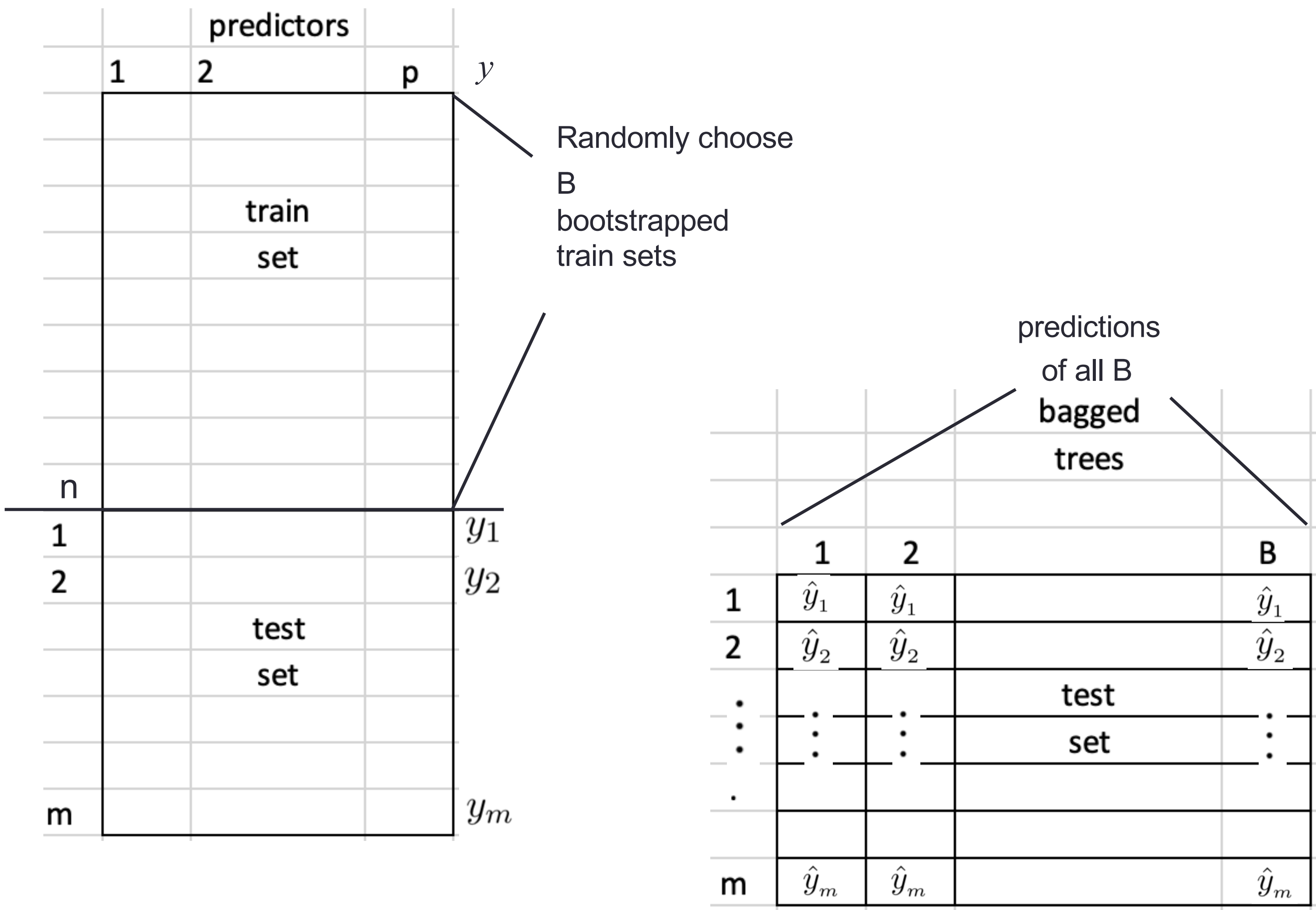
Bagging for Regression Trees



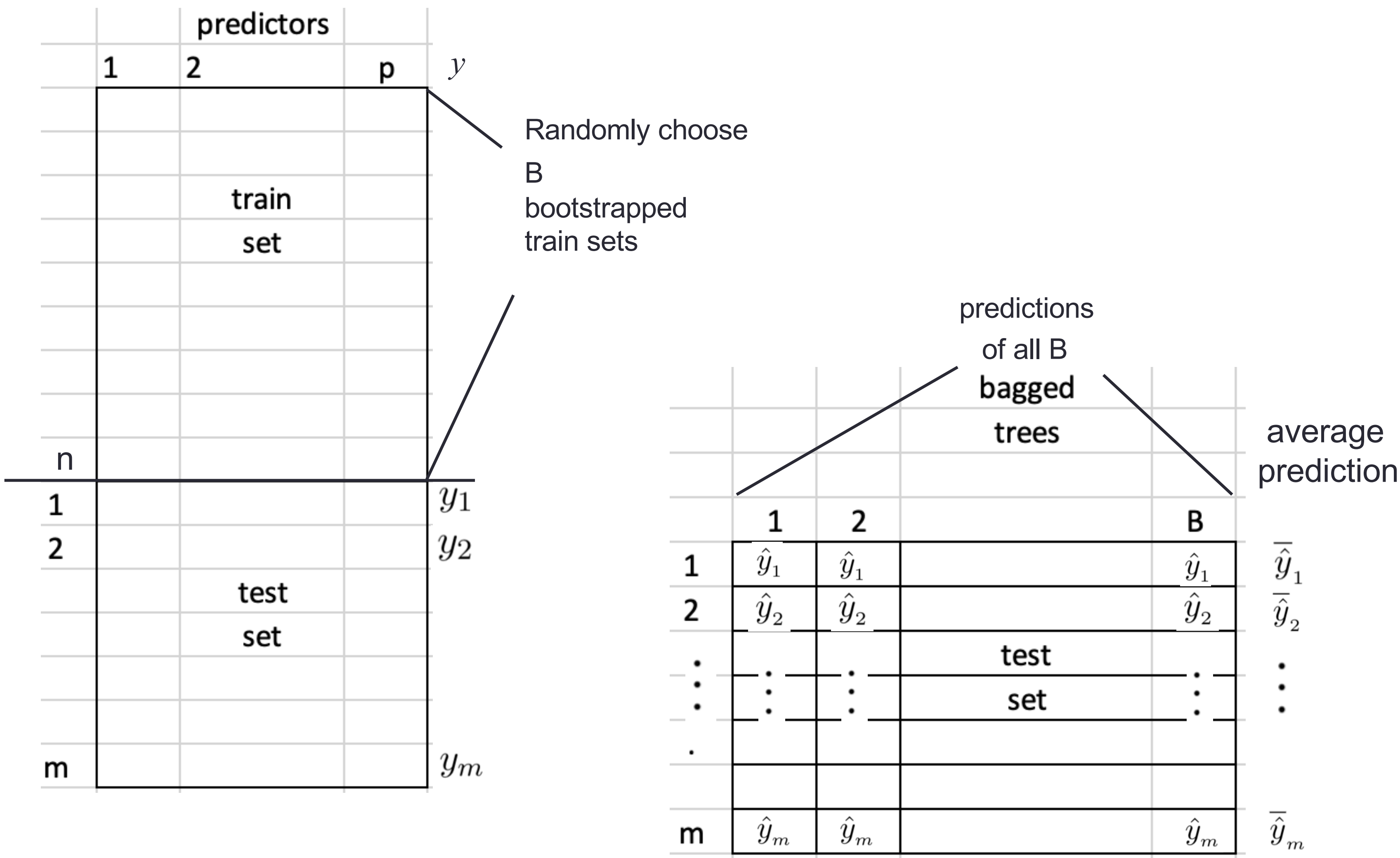
Bagging for Regression Trees



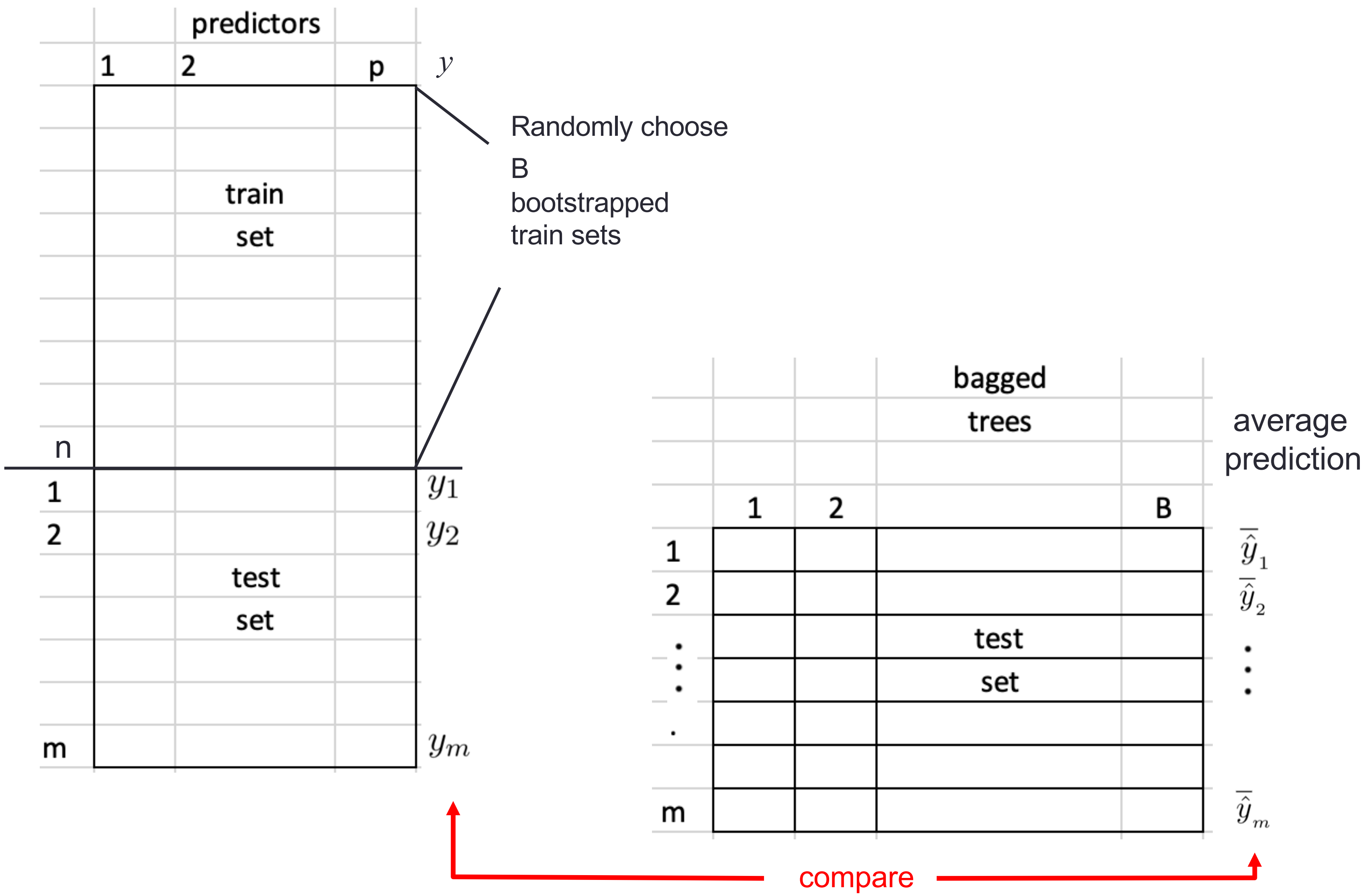
Bagging for Regression Trees



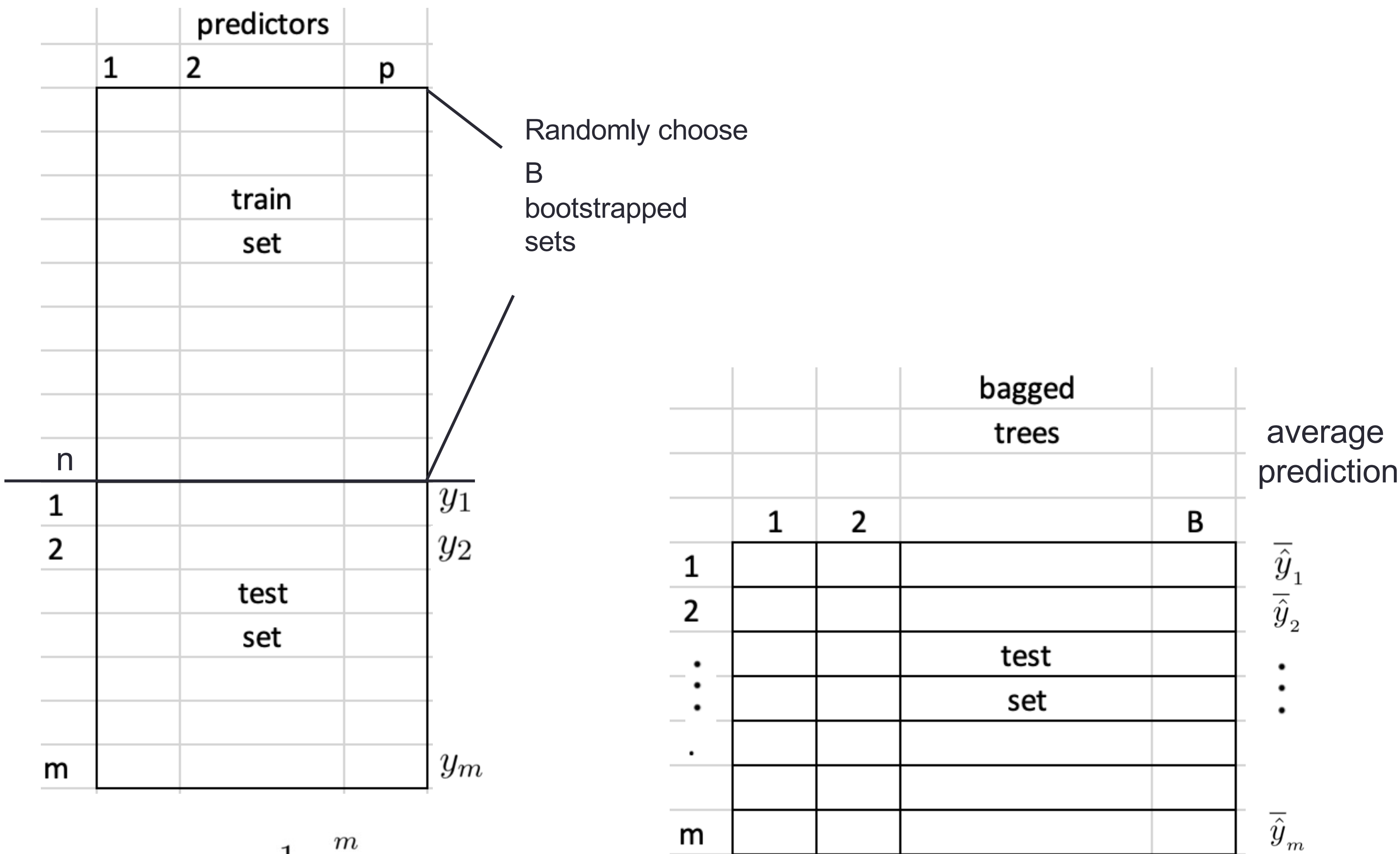
Bagging for Regression Trees



Bagging for Regression Trees



Bagging for Regression Trees



$$\text{MSPE} = \frac{1}{m}\sum_{i=1}^{m}(y_i - \bar{\hat{y}}_i)^2$$

Bagging - Notes

- Sometimes a few predictors are very good while many are poor predictors
- If so, many of the trees may contain the same set of powerful predictors
- Then the trees would yield similar predictions
- We say that the predictions are co-rrelated
- We need a way to de-correlate them

RANDOM FORESTS

Random Forests

It is a simple modification on bagging

How does it work?

- Before each split, randomly select a subset with m of the p predictors as candidates to make the split
- then choose the predictor giving the largest MSE reduction

Random Forests

It is a simple modification on bagging

How does it work?

- Before each split, randomly select a subset with m of the p predictors as candidates to make the split
- then choose the predictor giving the largest MSE reduction

- Bagging uses $m = p$ (all the predictors)
- Random Forest $m < p$ predictors

Why are we selecting *m* predictors instead of all *p* predictors for splitting?

- If there is a single strong predictor, most bagged trees will choose it for the first split (and for the following splits too)
- Most trees will look similar
- As a result their predictions will be highly correlated
- Averaging many highly correlated quantities does not lead to a large variance reduction
- By selecting the predictors for splits, from different subsets of predictors, Random Forest “de-correlates” the bagged trees leading to a reduction in variance

GRADIENT BOOSTING

Gradient Boosting

- Trees are built sequentially to improve upon the errors made by their predecessor trees
- Each new tree fits the data to the error made by the previous tree, predicting that error
- The new prediction is equal to the prediction of the previous tree plus α times the predicted error
- Parameter $0 < \alpha < 1$ is called the learning rate

Example 1 – Ensembles on Regression

Example – Boston dataset

Data of 506 houses in the area of Boston

- Want to predict the price of houses and to identify which variables are most important for prediction
- Split the dataset into a training (50%) and a test set
- Fit and compare bagged trees with 25 and 500 trees. Find the test MSPE.
- Fit a Random Forest with 500 trees and $\text{max\_features} = 6$. Which predictors are most important?
- Fit 500 Gradient boosted trees with $\text{max\_depth} = 4$, and $\alpha = 0.01, 0.20$. Which predictors are most important?

Boston dataset -13 features, 1 target

- CRIM - per capita crime rate by town
- ZN - proportion of residential land zoned for lots over 25,000 sq.ft.
- INDUS - proportion of non-retail business acres per town.
- CHAS - Charles River dummy variable (1 if tract bounds river; 0 otherwise)
- NOX - nitric oxides concentration (parts per 10 million)
- RM - average number of rooms per dwelling
- AGE - proportion of owner-occupied units built prior to 1940
- DIS - weighted distances to five Boston employment centres
- RAD - index of accessibility to radial highways
- TAX - full-value property-tax rate per $10,000
- PTRATIO - pupil-teacher ratio by town
- BLACK - proportion of blacks by town
- LSTAT - % lower status of the population
- MEDV - Median price of owner-occupied homes in $1000's

Example – libraries

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
```

```
from sklearn.model_selection import train_test_split
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import mean_squared_error
```

```
from sklearn.tree import DecisionTreeRegressor
from sklearn.ensemble import GradientBoostingRegressor
from sklearn.ensemble import RandomForestRegressor
```

Use RandomForestRegressor for both Bagging and Random Forest

Example – Boston dataset variables

```
boston_df = pd.read_csv('Boston.csv')
boston_df[:5]
```

	crim	zn	indus	chas	nox	rm	age	dis	rad	tax	ptratio	black	lstat	medv
0	0.00632	18.0	2.31	0	0.538	6.575	65.2	4.0900	1	296	15.3	396.90	4.98	24.0
1	0.02731	0.0	7.07	0	0.469	6.421	78.9	4.9671	2	242	17.8	396.90	9.14	21.6
2	0.02729	0.0	7.07	0	0.469	7.185	61.1	4.9671	2	242	17.8	392.83	4.03	34.7
3	0.03237	0.0	2.18	0	0.458	6.998	45.8	6.0622	3	222	18.7	394.63	2.94	33.4
4	0.06905	0.0	2.18	0	0.458	7.147	54.2	6.0622	3	222	18.7	396.90	5.33	36.2

```
X = boston_df.drop('medv',axis =1)
y = boston_df.medv
```

```
X_train,X_test,y_train,y_test = train_test_split(X,y,train_size=0.5,
                                                 random_state=0)
```

Ensembles on Regression – Bagging

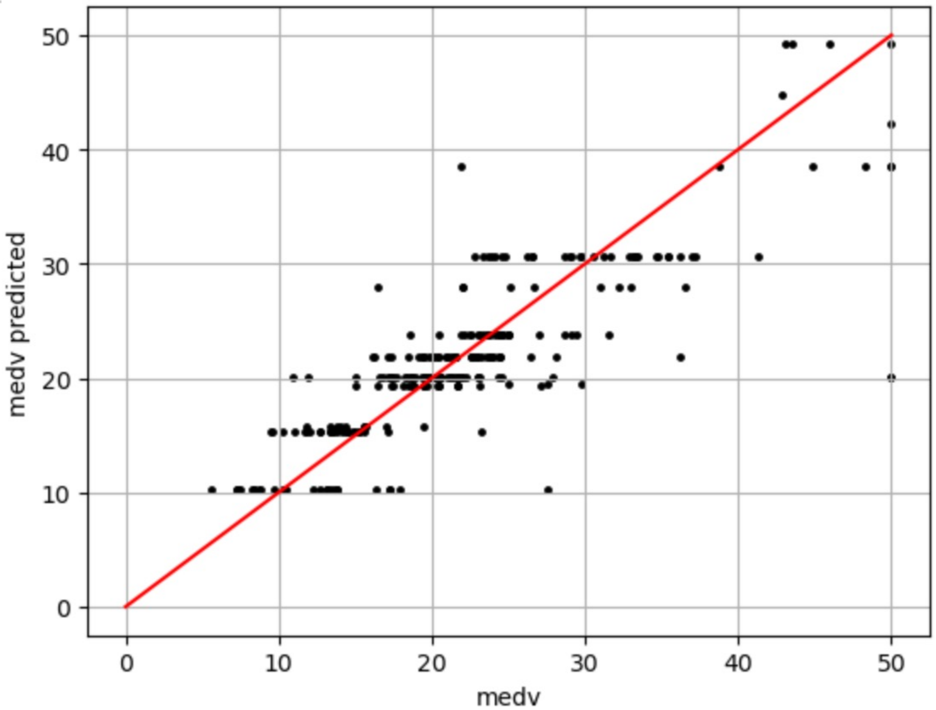
Boston dataset – Single tree

```
X_train,X_test,y_train,y_test =\
train_test_split(X,y,train_size=0.5,random_state=0)
```

```
tree1 = DecisionTreeRegressor(max_depth=4)
tree1.fit(X_train,y_train)
pred1 = tree1.predict(X_test)
mspe = mean_squared_error(y_test,pred1)
mspe
```

```
23.817371513828622
```

```
xaxis = np.linspace(0,50,100)
plt.scatter(y_test,pred1,s = 6,color='k')
plt.plot(xaxis,xaxis,color='r')
plt.xlabel('medv')
plt.ylabel('medv predicted')
```

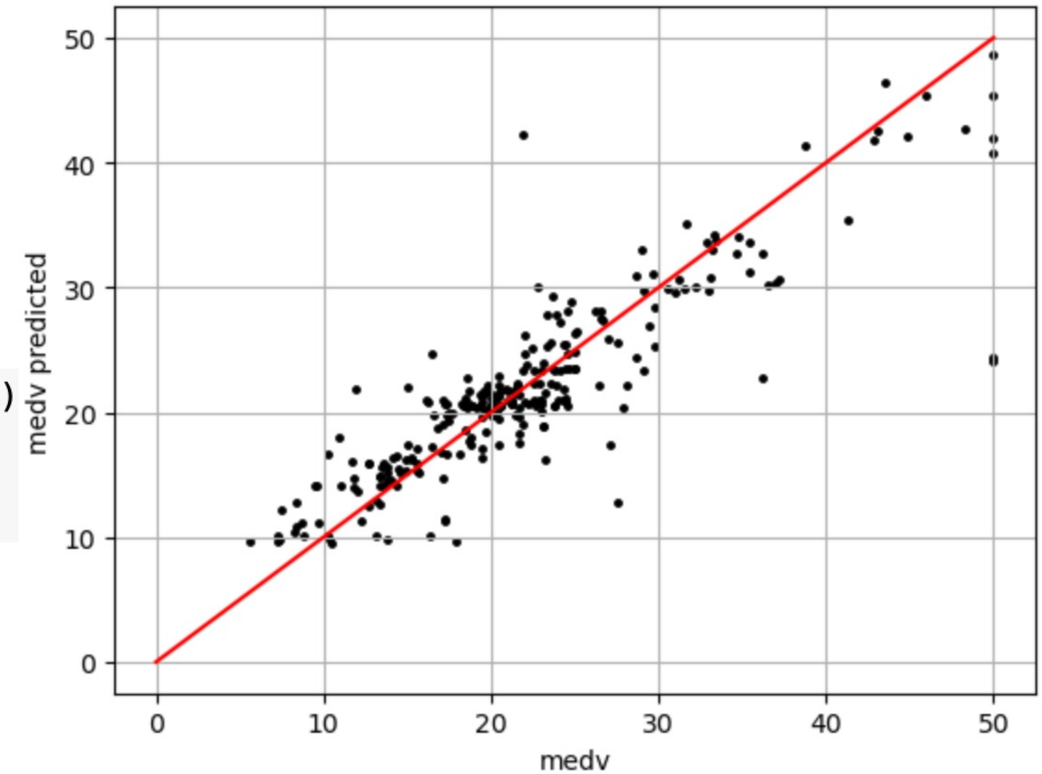


Boston dataset – Bagging 500 trees

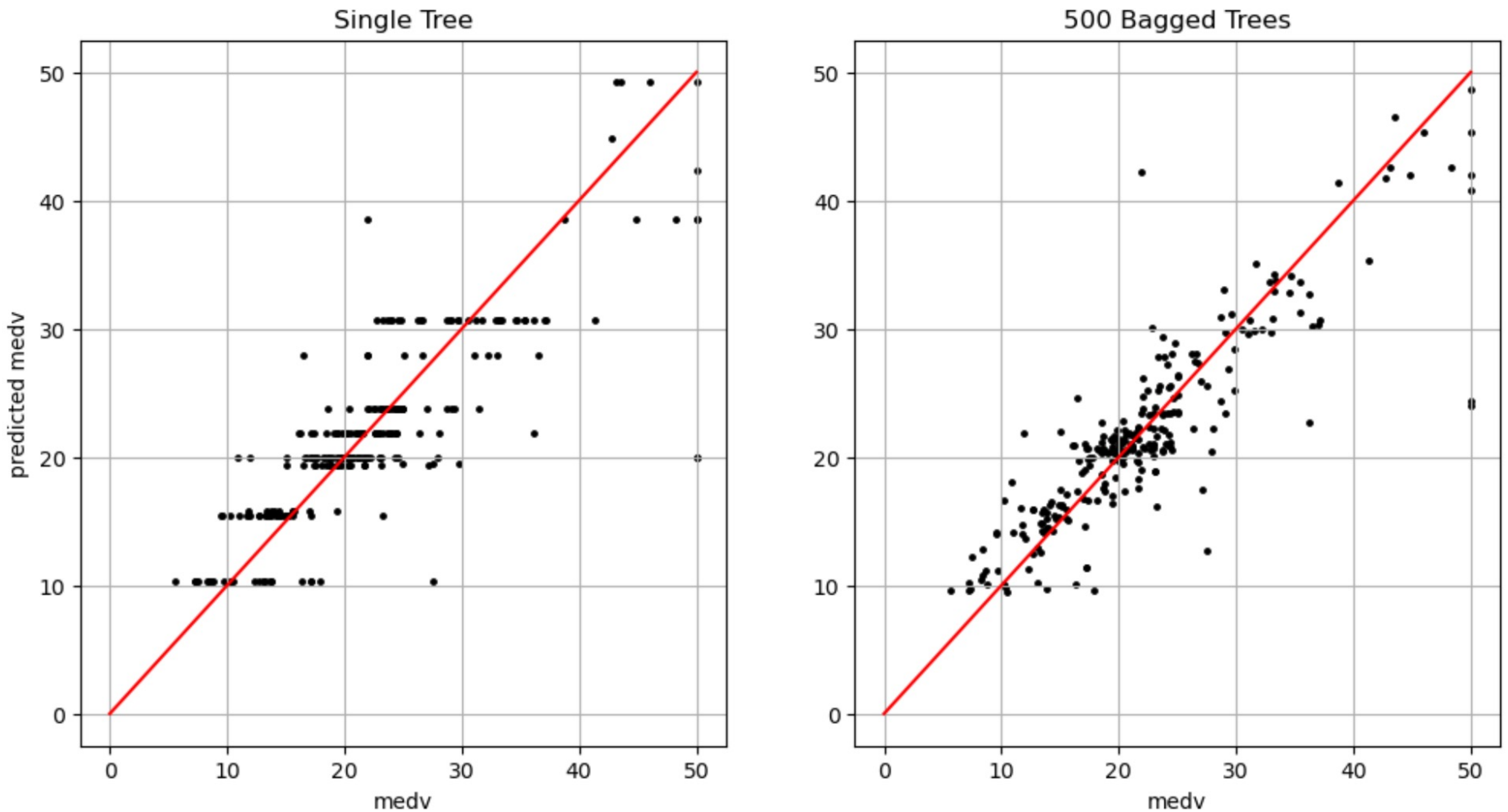
```
bag500 = RandomForestRegressor(max_features=13,max_depth=4,
                               n_estimators = 500,random_state=1)
bag500.fit(X_train,y_train);
pred2 = bag500.predict(X_test)
mean_squared_error(y_test,pred2)
```

17.511475948091437

```
plt.scatter(y_test,pred2,s = 6,color='k')
plt.plot(xaxis,xaxis,color='r')
plt.xlabel('medv')
plt.ylabel('medv predicted')
```



Single tree vs. Bagging 500 trees



Holdout CV – Finding best n_estimators

```
X_nontest,X_test,y_nontest,y_test = train_test_split(X,y,test_size=0.5,
                                                     random_state=0)
```

```
X_train,X_validation,y_train,y_validation = train_test_split(X_nontest,y_nontest,
                                                             train_size=0.5,
                                                             random_state=0)
```

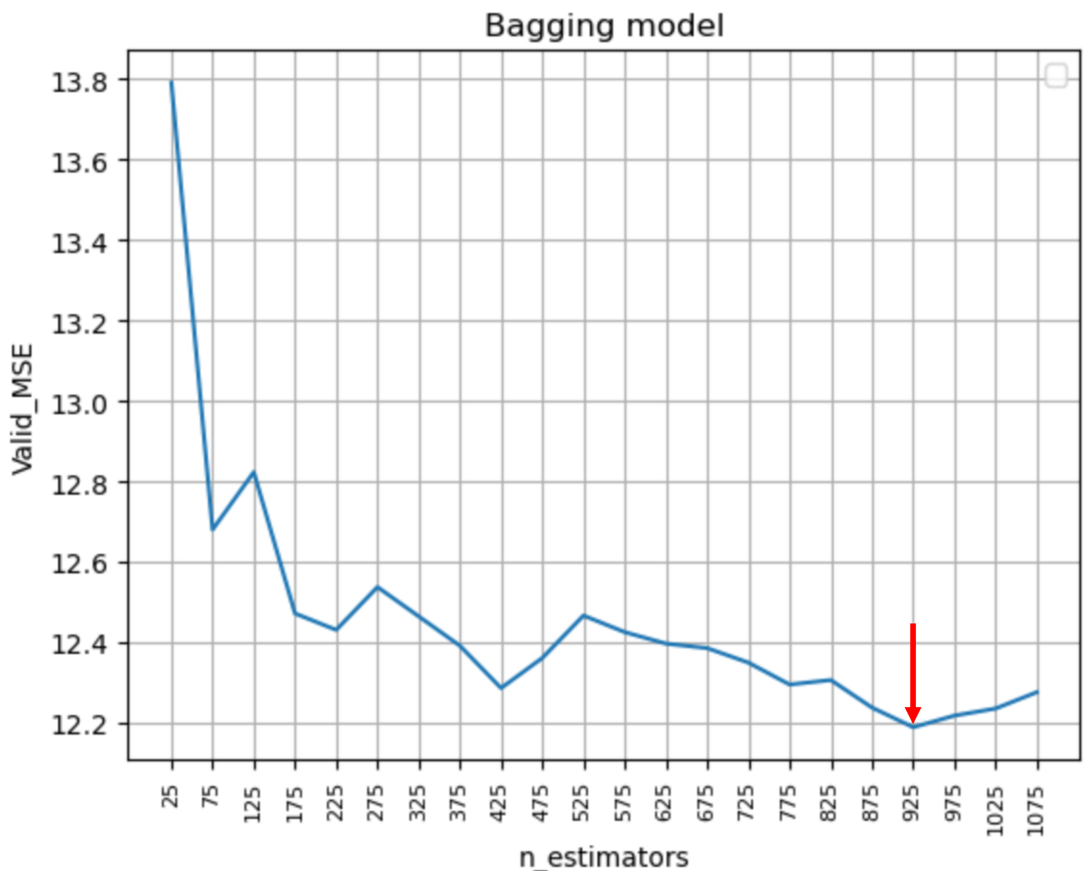
```
nn = range(25,1100,50)

mses = []
for k in nn:
    model = RandomForestRegressor(max_features=13,max_depth=4,
                                  n_estimators = k,
                                  random_state=1)
    model.fit(X_train,y_train);
    yhat = model.predict(X_validation)
    mse = mean_squared_error(y_validation,yhat)
    mses.append(mse)
```

try 25 <= n_estimators <= 1100

Holdout CV – Finding best n_estimators

```
df = pd.DataFrame(mses,columns = ['Valid_MSE'])
df.index = nn
df.index.name = 'n_estimators'
```

n_estimators	Valid_MSE
25	13.790862
75	12.680501
125	12.823402
175	12.472117
225	12.431220
275	12.537799
325	12.465000
375	12.392024
425	12.286661
475	12.361468
525	12.466939
575	12.425720
625	12.397227
675	12.385996
725	12.350061
775	12.295512
825	12.306713
875	12.238240
925	12.189376
975	12.218270
1025	12.235890
1075	12.276540



Holdout CV – Finding best n_estimators

```
df = pd.DataFrame(mses,columns = ['Valid_MSE'])
df.index = nn
df.index.name = 'n_estimators'
```

n_estimators	Valid_MSE
25	13.790862
75	12.680501
125	12.823402
175	12.472117
225	12.431220
275	12.537799
325	12.465000
375	12.392024
425	12.286661
475	12.361468
525	12.466939
575	12.425720
625	12.397227
675	12.385996
725	12.350061
775	12.295512
825	12.306713
875	12.238240
925	12.189376
975	12.218270
1025	12.235890
1075	12.276540

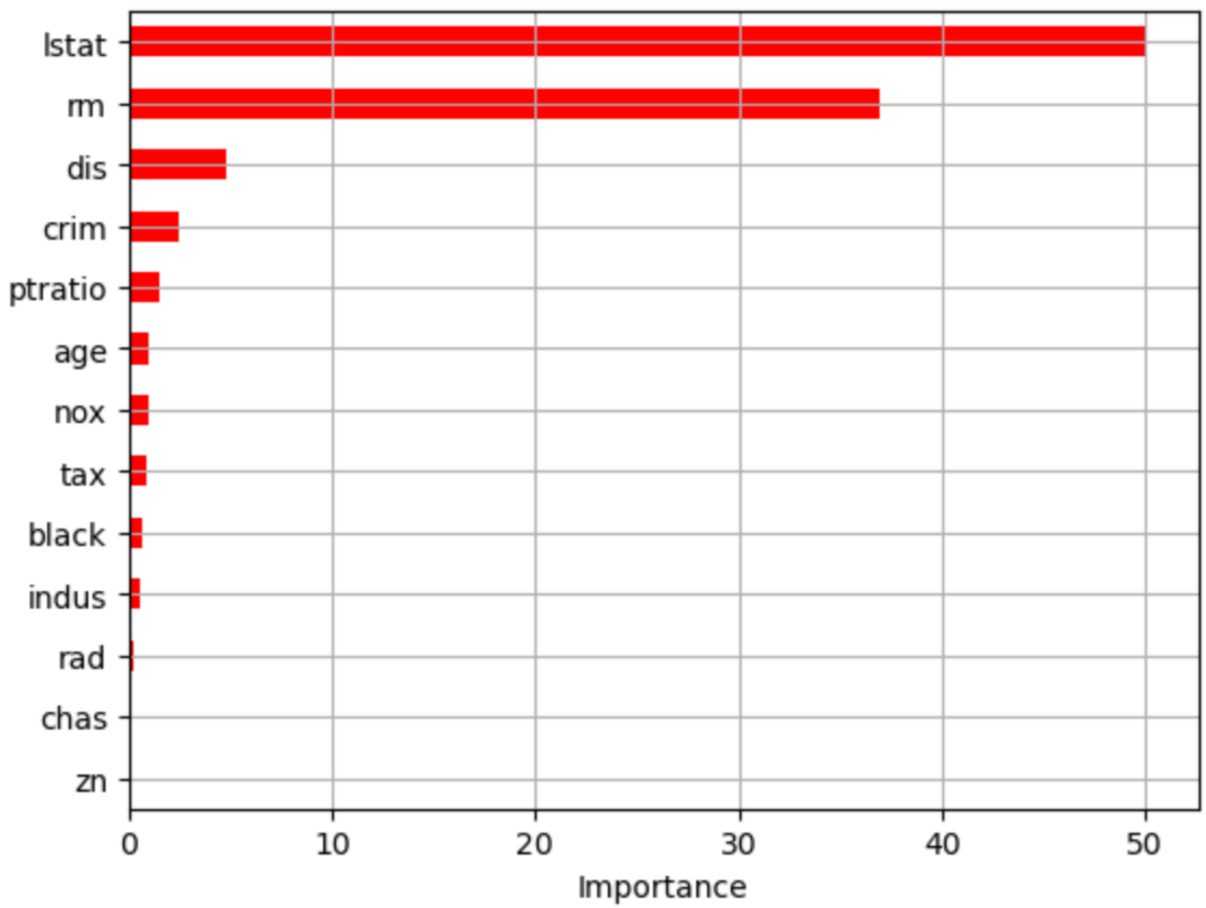
Test MSE

```
bagging_model =\
RandomForestRegressor(max_features=13,
                      max_depth=4,
                      n_estimators = 500,
                      random_state=1)
bagging_model.fit(X_nontest,y_nontest);
yhat = bagging_model.predict(X_test)
mean_squared_error(y_test,yhat)
```

```
17.511475948091437
```

Feature Importance – Bagging 500 trees

```
Importance = pd.DataFrame({'Importance':bagging_model.feature_importances_*100},
                          index = X.columns)
df8 = Importance.sort_values(by = 'Importance',axis = 0,
                             ascending = True)
df8.plot(kind = 'barh',color = 'r',legend = False)
```



The wealth level of the community ('lstat') and the house size ('rm') are the 2 most important predictors

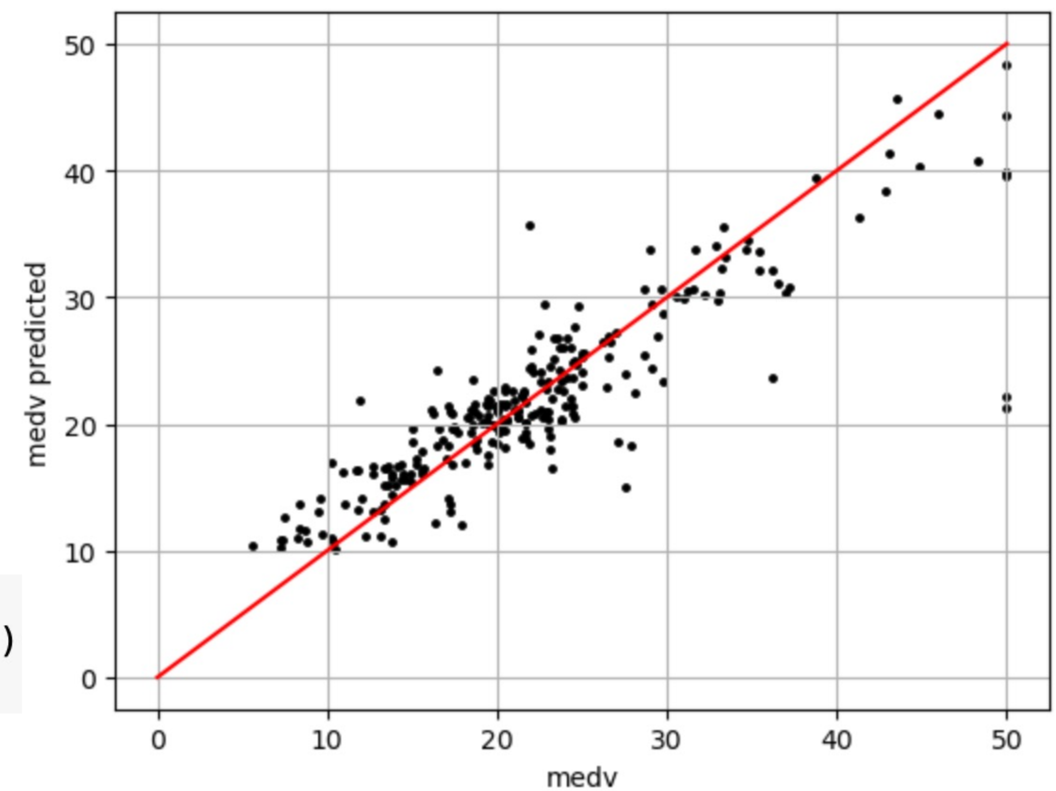
Ensembles on Regression – Random Forest

Random Forest with 500 trees

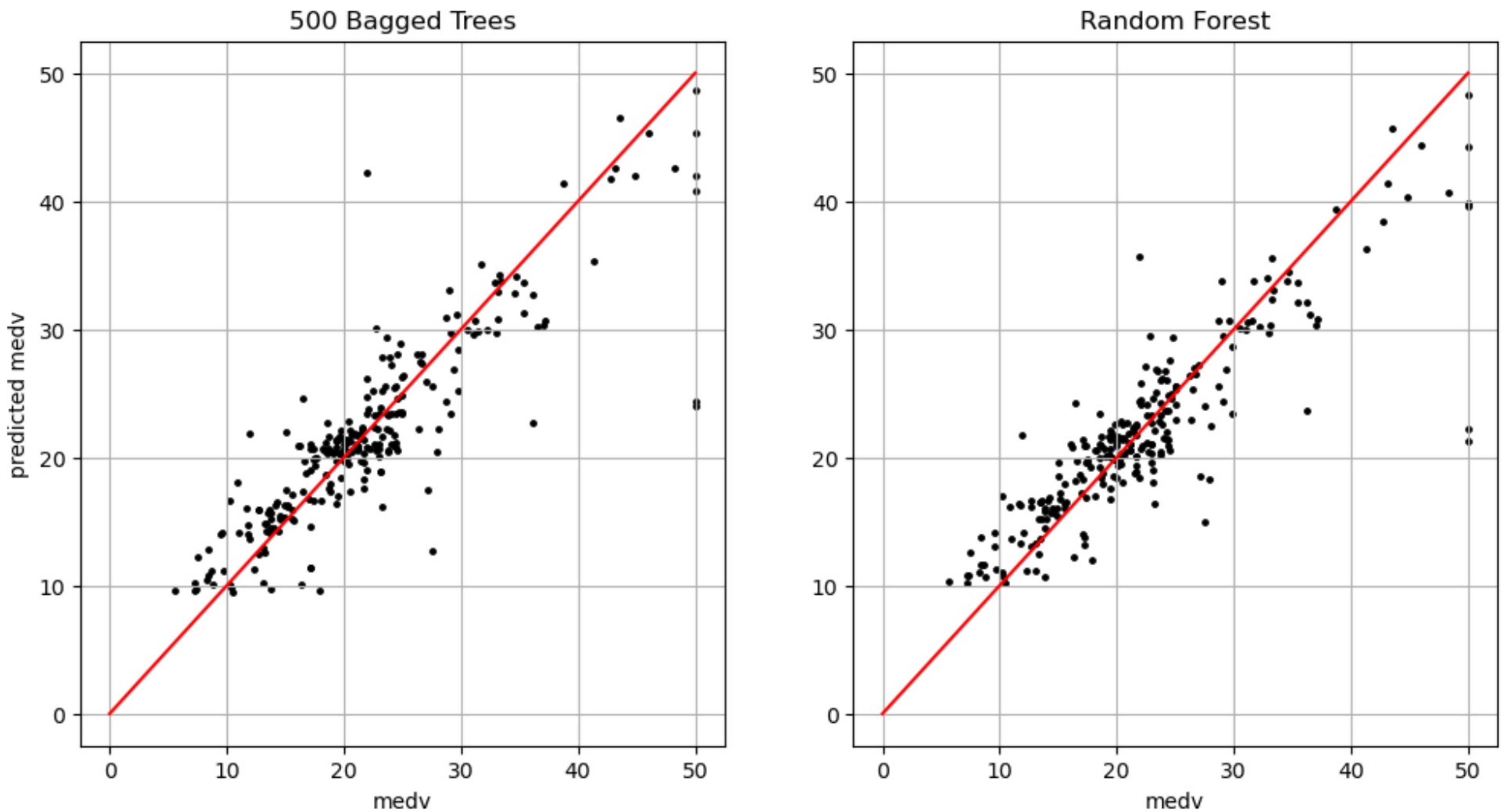
```
forest1 = RandomForestRegressor(max_features = 6,max_depth = 4,
                                n_estimators = 500,random_state = 1)
forest1.fit(X_train,y_train)
pred = forest1.predict(X_test)
mean_squared_error(y_test,pred)
```

```
17.434233348241854
```

```
xaxis = np.linspace(0,50,100)
plt.scatter(y_test,pred,s = 6,color='k')
plt.plot(xaxis,xaxis,color='r')
```



Bagging vs. Random Forest



Holdout CV – Finding best max_features

```
X_nontest,X_test,y_nontest,y_test = train_test_split(X,y,test_size=0.5,
                                                    random_state=0)
```

```
X_train,X_validation,y_train,y_validation = train_test_split(X_nontest,y_nontest,
                                                            train_size=0.5,
                                                            random_state=0)
```

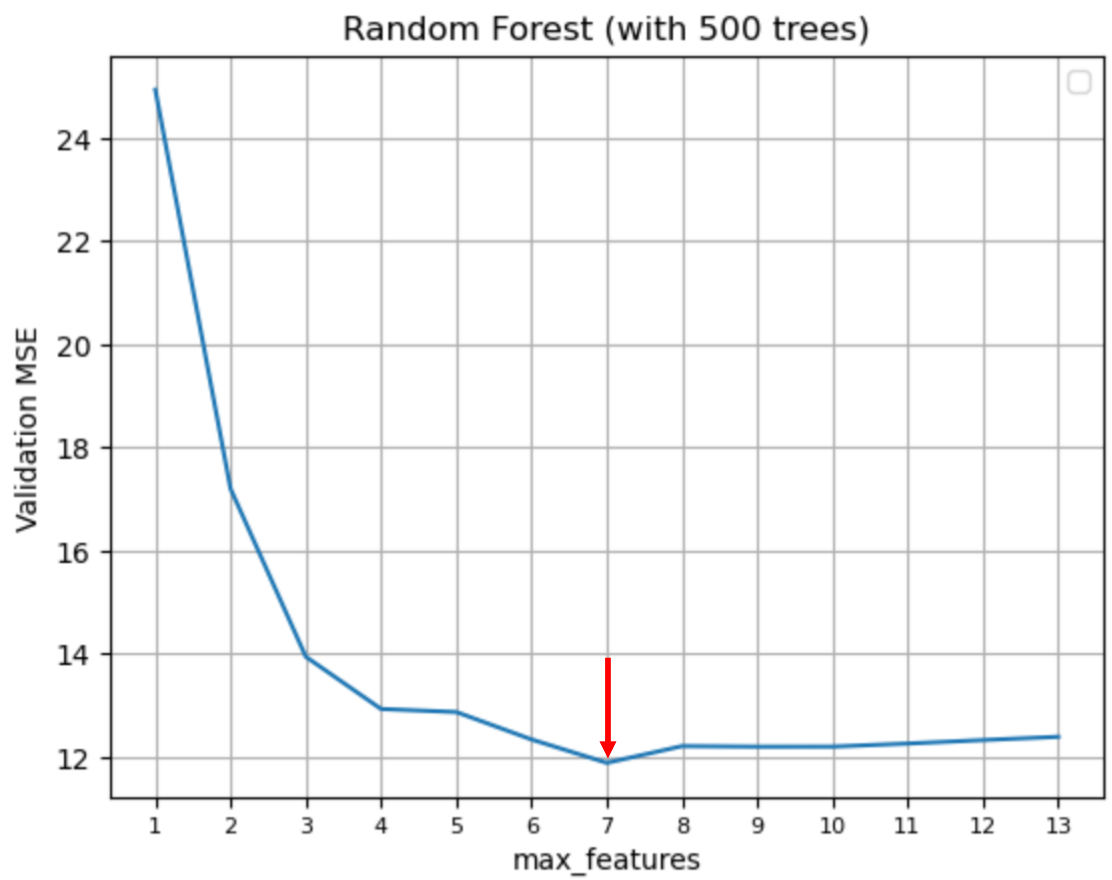
```
nn = range(1,14)

mses = []
for k in nn:
    model = RandomForestRegressor(max_features=k,max_depth=4,
                                  n_estimators = 500,
                                  random_state=1)
    model.fit(X_train,y_train)
    yhat = model.predict(X_validation)
    mse = mean_squared_error(y_validation,yhat)
    mses.append(mse)
```

try $1 \le m \le 13$

Holdout CV – Finding best max_features

```
df = pd.DataFrame(mses,columns = ['Valid_MSE'])
df.index = nn
df.index.name = 'max_features'
```

max_features	Valid_MSE
1	24.939713
2	17.200655
3	13.943997
4	12.930648
5	12.871667
6	12.339600
7	11.886289
8	12.210822
9	12.199773
10	12.200606
11	12.260297
12	12.328083
13	12.391338



Holdout CV – Finding best max_features

```
df = pd.DataFrame(mses,columns = ['Valid_MSE'])
df.index = nn
df.index.name = 'max_features'
```

max_features	Valid_MSE
1	24.939713
2	17.200655
3	13.943997
4	12.930648
5	12.871667
6	12.339600
7	11.886289
8	12.210822
9	12.199773
10	12.200606
11	12.260297
12	12.328083
13	12.391338

Test MSE

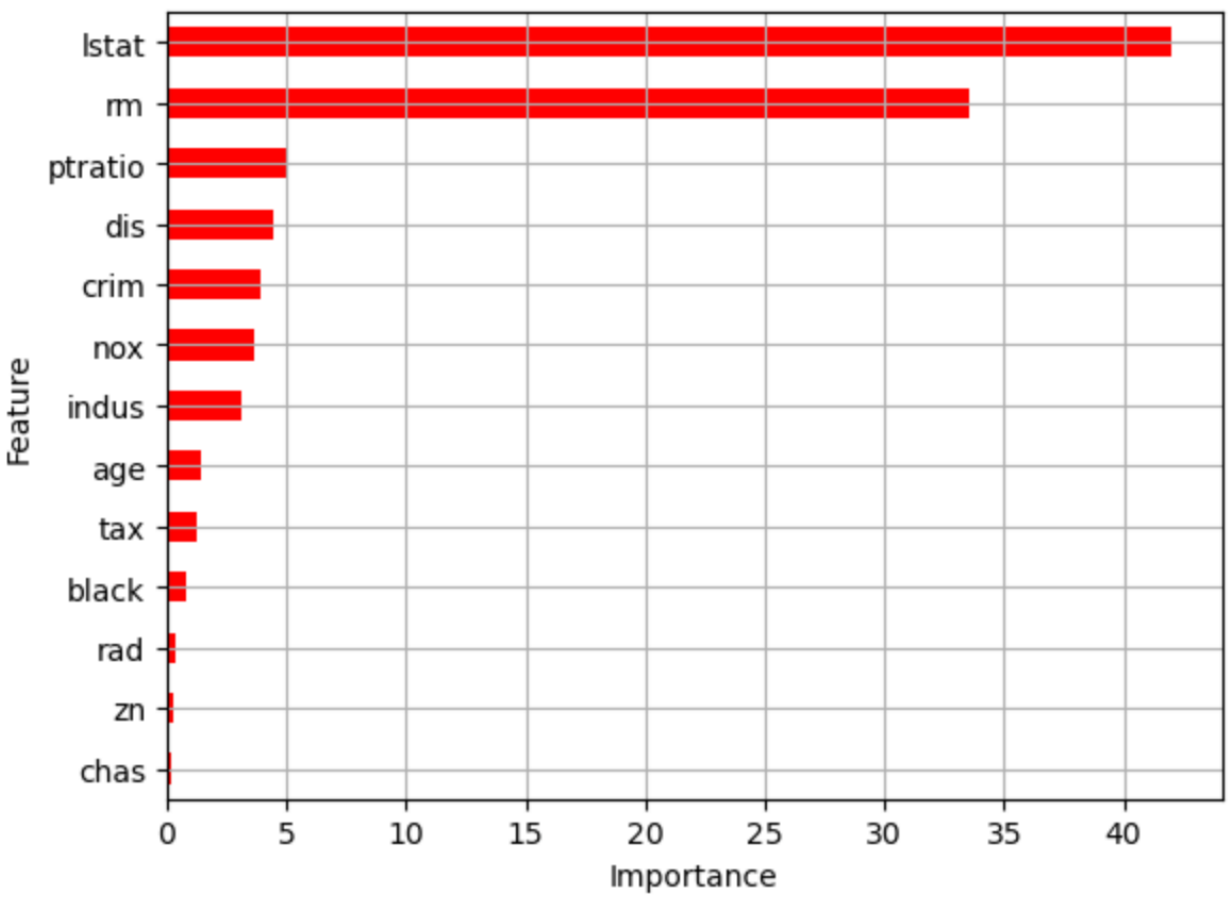
```
RF_best_model = RandomForestRegressor(max_features=7,
                                      max_depth=4,
                                      n_estimators = 500,
                                      random_state=1)
RF_best_model.fit(X_nontest,y_nontest);
yhat = RF_best_model.predict(X_test)
mean_squared_error(y_test,yhat)
```

```
17.31901376537634
```

```
# similar to Bagging Test MSE
```

Feature Importance – Random Forest

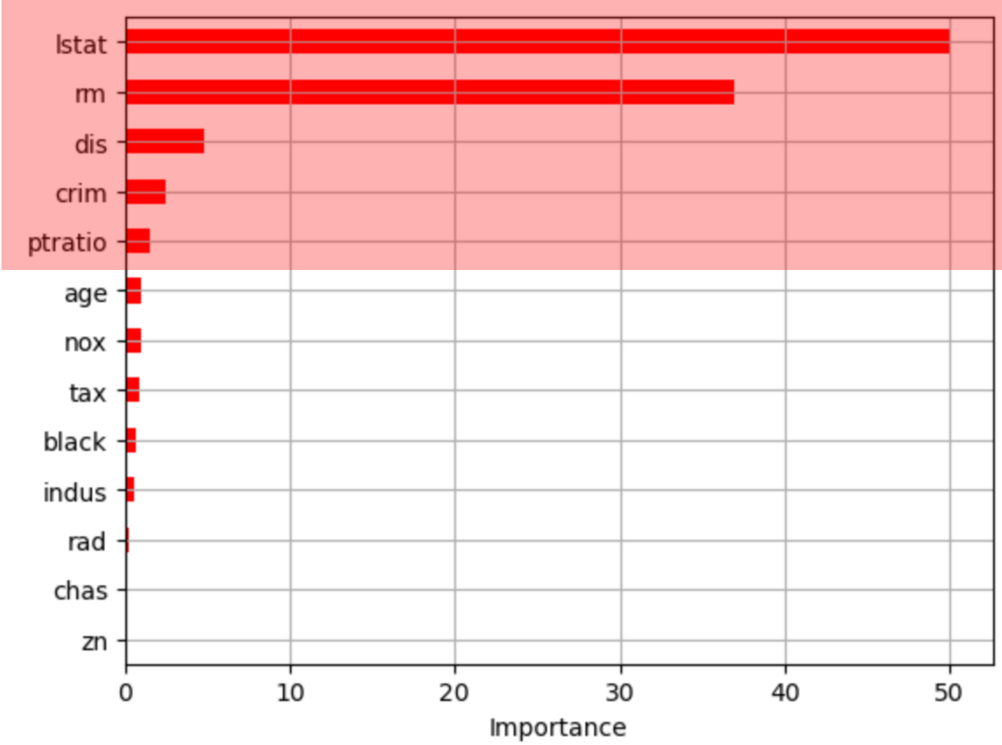
```
Importance = pd.DataFrame({'Importance':RF_best_model.feature_importances_*100},
                          index = X.columns)
df8 = Importance.sort_values(by = 'Importance',axis = 0,
                             ascending = True)
df8.plot(kind = 'barh',color = 'r',legend = False)
```

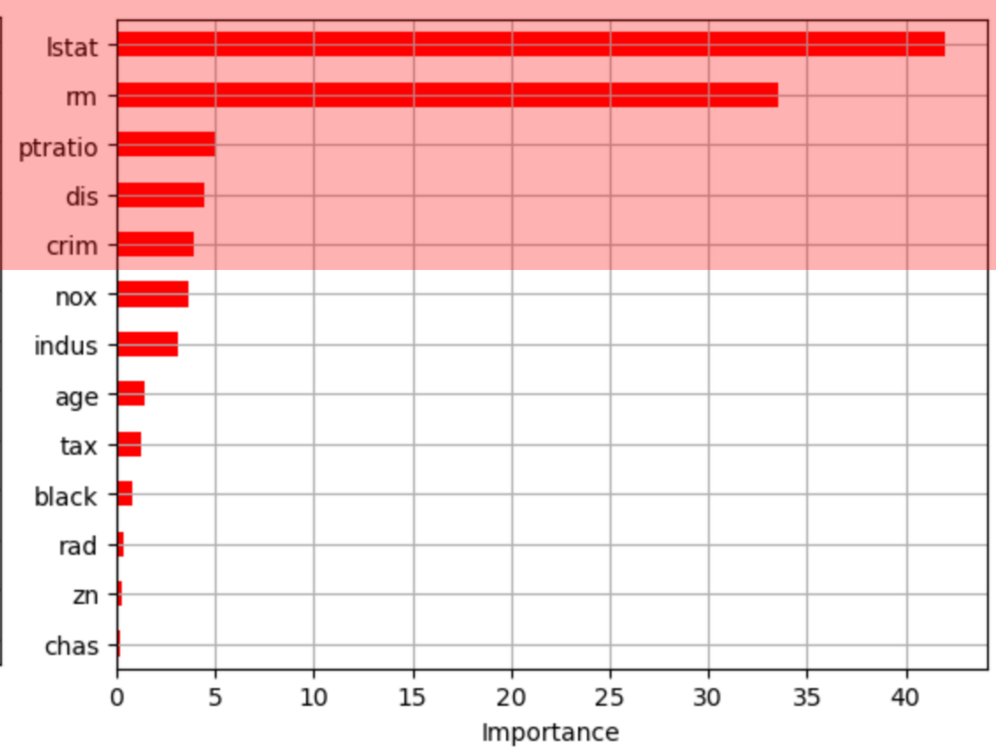


The wealth level of the community ('lstat') and the house size ('rm') are the 2 most important predictors

Feature Importance

Bagging **Random Forest**





Both models agree on the best 5 predictors

Ensembles on Regression – Gradient Boosting

Gradient Boosting Trees - learning_rate

learning_rate to 0.10

```
boost10 = GradientBoostingRegressor(n_estimators = 500,
                                    learning_rate = 0.1,
                                    max_depth = 4,
                                    random_state =1)
boost10.fit(X_train,y_train)
mean_squared_error(y_test,boost10.predict(X_test))
```

```
17.14218723022166
```

learning_rate to 0.40

```
boost40 = GradientBoostingRegressor(n_estimators = 500,
                                    learning_rate = 0.40,
                                    max_depth = 4,
                                    random_state=1)
boost40.fit(X_train,y_train)
mean_squared_error(y_test,boost40.predict(X_test))
```

```
16.580039404288993
```

GridSearchCV on the learning_rate

```
X_train,X_test,y_train,y_test = train_test_split(X,y,train_size=0.5,
                                                 random_state=0)
```

```
lrates = np.linspace(0.01,1,20)
lrates
```

← try these learning rates

```
array([0.01      , 0.06210526, 0.11421053, 0.16631579, 0.21842105,
       0.27052632, 0.32263158, 0.37473684, 0.42684211, 0.47894737,
       0.53105263, 0.58315789, 0.63526316, 0.68736842, 0.73947368,
       0.79157895, 0.84368421, 0.89578947, 0.94789474, 1.        ])
```

```
params = dict(learning_rate = lrates)
params
```

use the sklearn parameter name

```
{'learning_rate': array([0.01      , 0.06210526, 0.11421053, 0.16631579, 0.21842105,
        0.27052632, 0.32263158, 0.37473684, 0.42684211, 0.47894737,
        0.53105263, 0.58315789, 0.63526316, 0.68736842, 0.73947368,
        0.79157895, 0.84368421, 0.89578947, 0.94789474, 1.        ])}
```

GridSearchCV on the learning_rate

.

```
model = GradientBoostingRegressor(n_estimators = 500,
                                  max_depth = 4,
                                  random_state=1)
```

```
grid1 = GridSearchCV(model,param_grid = params,
                     scoring = 'neg_mean_squared_error',cv = 5)
grid1.fit(X_train,y_train);
```

```
grid1.best_params_
```

```
{'learning_rate': 0.1142105263157894 7}
```

```
-grid1.score(X_test,y_test)
```

← Test MSE

```
17.52561491676905 4
```

GridSearchCV on the learning_rate

```
# Refine params values

lrates = np.linspace(0.05,0.15,20)
params = dict(learning_rate = lrates)
lrates
```

← now try these learning rates

```
array([0.05      , 0.05526316, 0.06052632, 0.06578947, 0.07105263,
       0.07631579, 0.08157895, 0.08684211, 0.09210526, 0.09736842,
       0.10263158, 0.10789474, 0.11315789, 0.11842105, 0.12368421,
       0.12894737, 0.13421053, 0.13947368, 0.14473684, 0.15      ])
```

```
grid2 = GridSearchCV(model,param_grid = params,
                     scoring = 'neg_mean_squared_error',cv = 5)
grid2.fit(X_train,y_train);
```

```
grid2.best_params_
```

```
{'learning_rate': 0.10263157894736842}
```

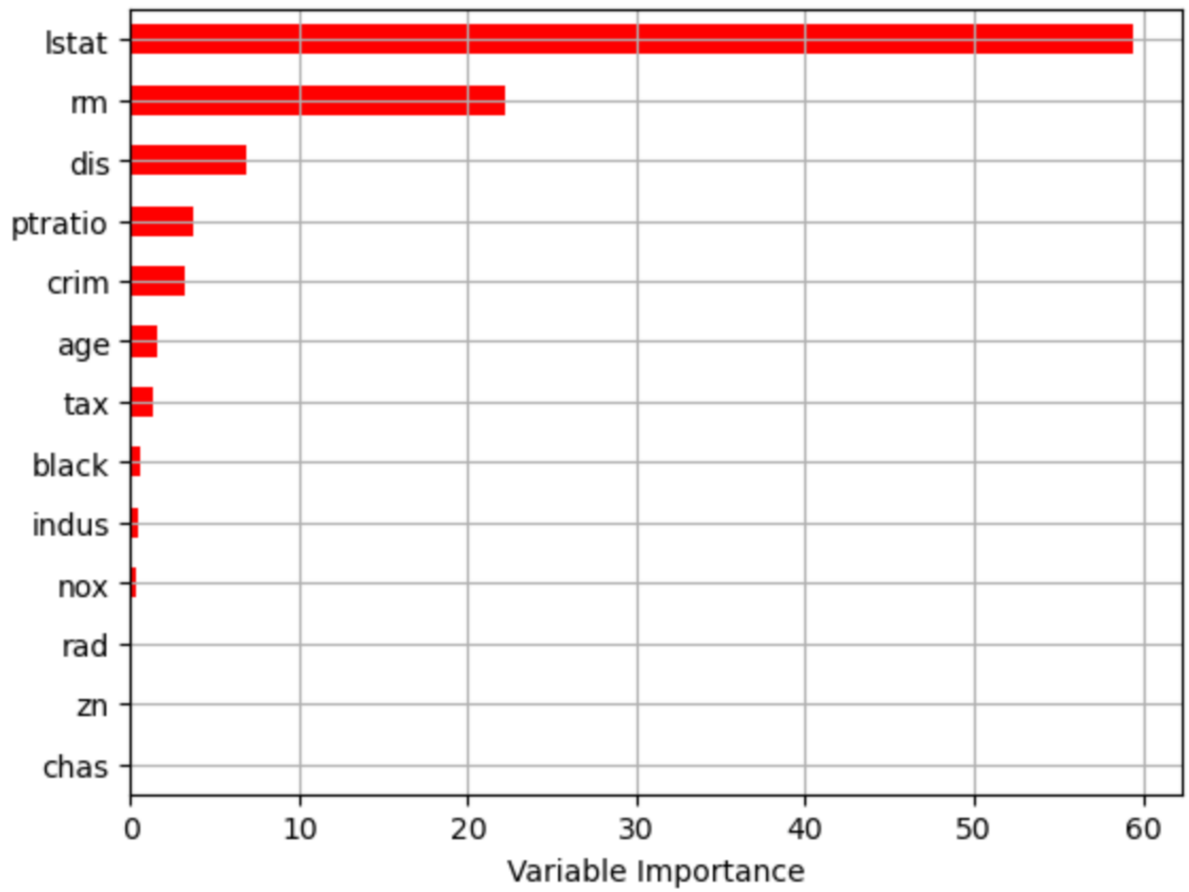
```
-grid2.score(X_test,y_test)
```

← Test MSE

```
16.879372503369037
```

Feature Importance – Gradient Boosting

```
Importance2 = grid2.best_estimator_.feature_importances_*100
Importance2 = pd.DataFrame({'Importance':Importance2},index = X.columns)
df9 = Importance2.sort_values(by = 'Importance',axis = 0,ascending = True)
df9.plot(kind = 'barh',color = 'r',legend = False)
```



The wealth level of the community ('lstat') and the house size ('rm') are the 2 most important predictors

GridSearchCV - Tuning 2 hyperparameters

GridSearchCV on learning_rate and max_features

```
# Consider 6 values for each parameter
params = {'learning_rate': np.linspace(0.01,1,6),
              'max_features': list(range(3,9))}
params
```

```
{'learning_rate': array([0.01 , 0.208, 0.406, 0.604, 0.802, 1.   ]),
 'max_features': [3, 4, 5, 6, 7, 8]}
```

```
model = GradientBoostingRegressor(n_estimators = 500,
                                  max_depth = 4,
                                  random_state=1)
```

```
grid1 = GridSearchCV(model,param_grid = params,
                    scoring = 'neg_mean_squared_error',cv = 5)
grid1.fit(X_train,y_train);
```

```
grid1.best_params_
```

```
{'learning_rate': 0.20800000000000002, 'max_features': 6}
```

GridSearchCV - Tuning 2 hyperparameters

```
df1 = pd.DataFrame(grid1.cv_results_)
```

		4	5	...	12		
mean_score_time	std_score_time	param_learning_rate	param_max_features	...	mean_test_score	std_test_score	rank_test_score
0.001017	0.000075	0.01	3		-11.615787	4.872017	6
0.001019	0.000030	0.01	4		-11.435279	5.473211	3
0.001120	0.000120	0.01	5		-11.159788	4.803937	2
0.001040	0.000090	0.01	6		-11.556346	5.215419	4
0.000947	0.000008	0.01	7		-11.891890	5.843181	7
0.001069	0.000097	0.01	8		-12.263327	6.333880	9
0.000904	0.000010	0.208	3	...	-12.377763	7.018744	11
⋮							⋮

↑ Validation MSE

GridSearchCV - Tuning 2 hyperparameters

```
params
```

```
{'learning_rate': array([0.01 , 0.208, 0.406, 0.604, 0.802, 1.   ]),
 'max_features': [3, 4, 5, 6, 7, 8]}
```

```
df2 = df1.iloc[:,[4,5,12]].copy()
df2.mean_test_score = -df2.mean_test_score
df2[:9]
```

	param_learning_rate	param_max_features	mean_test_score
0	0.01	3	11.615787
1	0.01	4	11.435279
2	0.01	5	11.159788
3	0.01	6	11.556346
4	0.01	7	11.891890
5	0.01	8	12.263327
6	0.208	3	12.377763
7	0.208	4	13.334128
8	0.208	5	12.239297

```
df2[-9:]
```

	param_learning_rate	param_max_features	mean_test_score
27	0.802	6	14.094091
28	0.802	7	20.887565
29	0.802	8	19.463552
30	1.000	3	37.093306
31	1.000	4	46.481536
32	1.000	5	41.303353
33	1.000	6	19.580221
34	1.000	7	29.236760
35	1.000	8	25.724720

GridSearchCV - Tuning 2 hyperparameters

```
df2.param_learning_rate = df2.param_learning_rate.astype('float64')
df2.param_learning_rate = df2.param_learning_rate.round(3)
```

```
df2 = df2.pivot_table('mean_test_score',
                      columns = 'param_learning_rate',
                      index = 'param_max_features')
df2
```

param_learning_rate	**0.010**	**0.208**	**0.406**	**0.604**	**0.802**	**1.000**
param_max_features						
3	11.615787	12.377763	17.180822	20.829473	30.899405	37.093306
4	11.435279	13.334128	16.238018	24.198933	31.339964	46.481536
5	11.159788	12.239297	18.045442	24.659544	34.942586	41.303353
6	11.556346	11.070416	11.565877	13.408052	14.094091	19.580221
7	11.891890	12.363371	13.125690	17.740055	20.887565	29.236760
8	12.263327	13.577059	13.556925	15.655501	19.463552	25.724720

GridSearchCV - Tuning 2 hyperparameters

```
df2.param_learning_rate = df2.param_learning_rate.astype('float64')
df2.param_learning_rate = df2.param_learning_rate.round(3)
```

```
df2 = df2.pivot_table('mean_test_score',
                      columns = 'param_learning_rate',
                      index = 'param_max_features')
df2
```

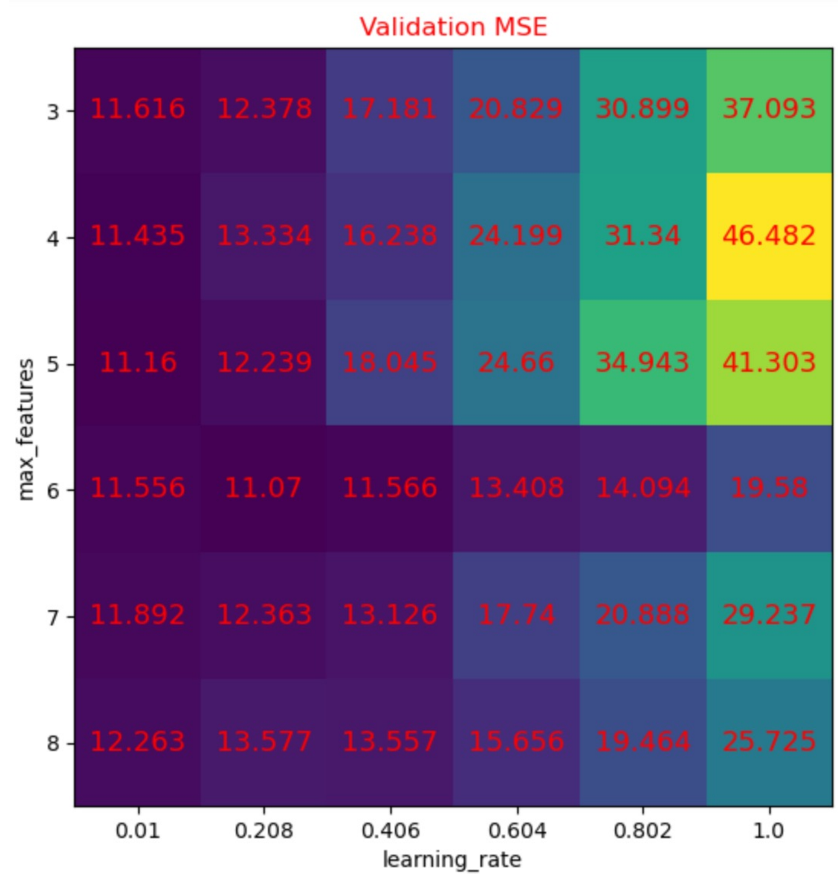
param_learning_rate	**0.010**	**0.208**	**0.406**	**0.604**	**0.802**	**1.000**
param_max_features						
3	11.615787	12.377763	17.180822	20.829473	30.899405	37.093306
4	11.435279	13.334128	16.238018	24.198933	31.339964	46.481536
5	11.159788	12.239297	18.045442	24.659544	34.942586	41.303353
6	11.556346	11.070416	11.565877	13.408052	14.094091	19.580221
7	11.891890	12.363371	13.125690	17.740055	20.887565	29.236760
8	12.263327	13.577059	13.556925	15.655501	19.463552	25.724720

GridSearchCV - Tuning 2 hyperparameters

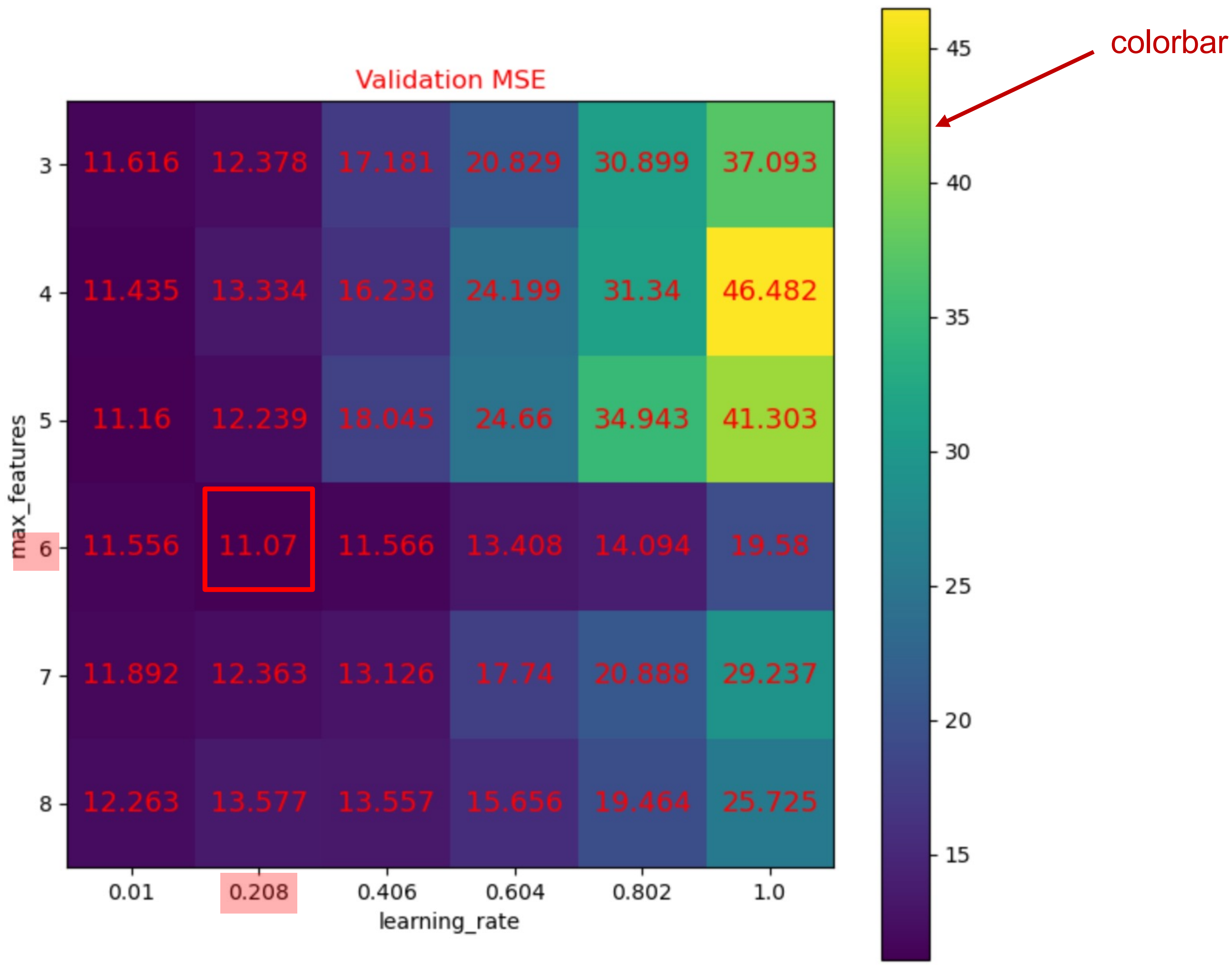
```
# transform dataframe to numpy array
arates = df2.values
arates = np.round(arates,3)
arates
```

```
array([[11.616, 12.378, 17.181, 20.829, 30.899, 37.093],
       [11.435, 13.334, 16.238, 24.199, 31.34 , 46.482],
       [11.16 , 12.239, 18.045, 24.66 , 34.943, 41.303],
       [11.556, 11.07 , 11.566, 13.408, 14.094, 19.58 ],
       [11.892, 12.363, 13.126, 17.74 , 20.888, 29.237],
       [12.263, 13.577, 13.557, 15.656, 19.464, 25.725]])
```

```
plt.figure(figsize=(8,8))
plt.xticks(range(6),df2.columns)
plt.yticks(range(6),df2.index)
plt.ylabel('max_features')
plt.xlabel('learning_rate')
plt.title('Validation MSE',c='r')
plt.imshow(arates)
for i in range(6):
    for j in range(6):
        text = plt.text(j,i,arates[i,j],
                        ha="center",
                        va="center",
                        color="r",
                        size = 13)
plt.colorbar();
```



GridSearchCV - Tuning 2 hyperparameters



GridSearchCV - Tuning 2 hyperparameters

Test MSE

```
model = GradientBoostingRegressor(n_estimators = 500,
                                  max_features = 6,
                                  max_depth = 4,
                 learning_rate = 0.20800000000000002,
                                  random_state=1)
model.fit(X_train,y_train)
pred2 = model.predict(X_test)
mean_squared_error(y_test,pred2)
```

```
15.688938784290462
```

```
-grid1.score(X_test,y_test)
```

```
15.688938784290462
```

GridSearchCV - Tuning 2 hyperparameters

Test MSE

```
model = GradientBoostingRegressor(n_estimators = 500,
                                  max_features = 6,
                                  max_depth = 4,
                learning_rate = 0.20800000000000002,
                                  random_state=1)
model.fit(X_train,y_train)
pred2 = model.predict(X_test)
mean_squared_error(y_test,pred2)
```

15.688938784290462

```
-grid1.score(X_test,y_test)
```

15.688938784290462

```
model = GradientBoostingRegressor(n_estimators = 500,
                                  max_features = 6,
                                  max_depth = 4,
                                  learning_rate = 0.208,
                                  random_state=1)
model.fit(X_train,y_train)
pred2 = model.predict(X_test)
mean_squared_error(y_test,pred2)
```

15.127755055051313 ← Best Test MSE