

# **Data Science III with python (Class notes)**

**STAT 303-3**

Lizhen Shi

2025-04-02

# Table of contents

<b>Preface</b>	<b>7</b>
<b>I Bias &amp; Variance; KNN</b>	<b>8</b>
<b>1 Bias-variance tradeoff</b>	<b>9</b>
1.1 Simple model (Less flexible) . . . . .	9
1.2 Complex model (more flexible) . . . . .	12
<b>2 KNN</b>	<b>15</b>
2.1 KNN for regression . . . . .	15
2.1.1 Scaling data . . . . .	16
2.1.2 Fitting and validating model . . . . .	16
2.1.3 Hyperparameter tuning . . . . .	17
2.1.4 KNN hyperparameters . . . . .	23
2.2 KNN for classification . . . . .	24
<b>3 Hyperparameter tuning</b>	<b>25</b>
3.1 GridSearchCV . . . . .	26
3.2 RandomizedSearchCV() . . . . .	29
3.3 BayesSearchCV() . . . . .	31
3.3.1 Diagonosis of cross-validated score optimization . . . . .	34
3.3.2 Live monitoring of cross-validated score . . . . .	39
3.4 cross_validate() . . . . .	40
<b>II Tree based models</b>	<b>44</b>
<b>4 Regression trees</b>	<b>45</b>
4.1 Building a regression tree . . . . .	46
4.2 Optimizing parameters to improve the regression tree . . . . .	49
4.2.1 Range of hyperparameter values . . . . .	49
4.2.2 Cross validation: Coarse grid . . . . .	49
4.2.3 Cross validation: Finer grid . . . . .	51
4.3 Cost complexity pruning . . . . .	53
4.3.1 Depth vs alpha; Node counts vs alpha . . . . .	56

4.3.2	Train and test accuracies (R-squared) vs alpha . . . . .	57
<b>5</b>	<b>Classification trees</b>	<b>59</b>
5.1	Building a classification tree . . . . .	60
5.2	Optimizing hyperparameters to optimize performance . . . . .	62
5.3	Optimizing the decision threshold probability . . . . .	64
5.3.1	Balancing recall with precision . . . . .	64
5.3.2	Balancing recall with false positive rate . . . . .	69
5.4	Cost complexity pruning . . . . .	73
<b>6</b>	<b>Bagging</b>	<b>75</b>
6.1	Bagging regression trees . . . . .	76
6.1.1	Model accuracy vs number of trees . . . . .	77
6.1.2	Optimizing bagging hyperparameters using grid search . . . . .	80
6.2	Bagging for classification . . . . .	81
6.2.1	Model accuracy vs number of trees . . . . .	83
6.2.2	Optimizing bagging hyperparameters using grid search . . . . .	87
6.2.3	Tuning the decision threshold probability . . . . .	88
<b>7</b>	<b>Bagging (addendum)</b>	<b>94</b>
7.1	Tree without tuning . . . . .	95
7.2	Performance of tree improves with tuning . . . . .	97
7.3	Bagging tuned trees . . . . .	97
7.4	Bagging untuned trees . . . . .	97
7.5	Tuning bagged model - OOB . . . . .	98
7.6	Tuning without k-fold cross-validation . . . . .	98
7.7	warm start . . . . .	102
7.8	Bagging KNN . . . . .	103
<b>8</b>	<b>Random Forest</b>	<b>106</b>
8.1	Random Forest for regression . . . . .	108
8.1.1	Model accuracy vs number of trees . . . . .	109
8.1.2	Tuning random forest . . . . .	113
8.2	Random forest for classification . . . . .	116
8.2.1	Model accuracy vs number of trees . . . . .	118
8.2.2	Tuning random forest . . . . .	120
8.3	Random forest vs Bagging . . . . .	122
	Tuning random forest . . . . .	124
<b>9</b>	<b>Adaptive Boosting</b>	<b>128</b>
9.1	Hyperparameters . . . . .	128
9.2	AdaBoost for regression . . . . .	129
9.2.1	Number of trees vs cross validation error . . . . .	129

9.2.2	Depth of tree vs cross validation error . . . . .	131
9.2.3	Learning rate vs cross validation error . . . . .	133
9.2.4	Tuning AdaBoost for regression . . . . .	136
9.3	AdaBoost for classification . . . . .	141
9.3.1	Number of trees vs cross validation accuracy . . . . .	141
9.3.2	Depth of each tree vs cross validation accuracy . . . . .	143
9.3.3	Learning rate vs cross validation accuracy . . . . .	145
9.3.4	Tuning AdaBoost Classifier hyperparameters . . . . .	147
9.3.5	Tuning the decision threshold probability . . . . .	148
<b>10</b>	<b>Gradient Boosting</b>	<b>152</b>
10.1	Hyperparameters . . . . .	152
10.2	Gradient boosting for regression . . . . .	153
10.2.1	Number of trees vs cross validation error . . . . .	153
10.2.2	Depth of tree vs cross validation error . . . . .	155
10.2.3	Learning rate vs cross validation error . . . . .	157
10.2.4	Subsampling vs cross validation error . . . . .	159
10.2.5	Maximum features vs cross-validation error . . . . .	161
10.2.6	Tuning Gradient boosting for regression . . . . .	163
10.2.7	Ensemble modeling (for regression models) . . . . .	167
10.3	Gradient boosting for classification . . . . .	167
10.3.1	Number of trees vs cross validation accuracy . . . . .	167
10.3.2	Depth of each tree vs cross validation accuracy . . . . .	169
10.3.3	Learning rate vs cross validation accuracy . . . . .	171
10.3.4	Tuning Gradient boosting Classifier . . . . .	173
10.4	Faster algorithms and tuning tips . . . . .	177
<b>11</b>	<b>XGBoost</b>	<b>178</b>
11.1	Hyperparameters . . . . .	178
11.2	XGBoost for regression . . . . .	180
11.2.1	Number of trees vs cross validation error . . . . .	180
11.2.2	Depth of tree vs cross validation error . . . . .	181
11.2.3	Learning rate vs cross validation error . . . . .	183
11.2.4	Regularization ( <b>reg_lambda</b> ) vs cross validation error . . . . .	185
11.2.5	Regularization ( <b>gamma</b> ) vs cross validation error . . . . .	188
11.2.6	Tuning XGboost regressor . . . . .	190
11.2.7	Early stopping with XGBoost . . . . .	193
11.3	XGBoost for classification . . . . .	195
11.3.1	Precision & recall vs <b>scale_pos_weight</b> . . . . .	200
<b>12</b>	<b>LightGBM and CatBoost</b>	<b>203</b>
12.1	LightGBM . . . . .	204
12.1.1	LightGBM for regression . . . . .	205

12.1.2	LightGBM vs XGBoost . . . . .	208
12.2	CatBoost . . . . .	209
12.2.1	CatBoost for regression . . . . .	210
12.2.2	Target encoding with CatBoost . . . . .	211
12.2.3	CatBoost vs XGBoost . . . . .	212
12.2.4	Tuning <code>CatBoostRegressor</code> . . . . .	213
12.2.5	Tuning Tips . . . . .	215
<b>13</b>	<b>Ensemble modeling</b>	<b>216</b>
13.1	Ensembling regression models . . . . .	218
13.1.1	Voting Regressor . . . . .	218
13.1.2	Stacking Regressor . . . . .	221
13.2	Ensembling classification models . . . . .	225
	AdaBoost . . . . .	226
	Gradient Boosting . . . . .	226
	XGBoost . . . . .	227
13.2.1	Voting classifier - hard voting . . . . .	228
13.2.2	Voting classifier - soft voting . . . . .	229
13.2.3	Stacking classifier . . . . .	229
13.2.4	Tuning all models simultaneously . . . . .	231
13.3	Ensembling models based on different sets of predictors . . . . .	232
	<b>Appendices</b>	<b>236</b>
<b>A</b>	<b>Assignment 1</b>	<b>236</b>
	Instructions . . . . .	236
A.1	1) Bias-Variance Trade-off for Regression ( <b>50 points</b> ) . . . . .	236
A.1.1	a) Define the True Relationship (Signal) . . . . .	237
A.1.2	b) Generate Test Set (No Noise) . . . . .	237
A.1.3	c) Initialize Results DataFrame . . . . .	238
A.1.4	d) Generate Training Sets (With Noise) . . . . .	238
A.1.5	Reminder: Computing MSE on the Noiseless Test Set . . . . .	239
A.1.6	e) Visualize Bias-Variance Tradeoff . . . . .	239
A.1.7	f) Identify the Optimal Model . . . . .	240
A.2	2) Building a Low-Bias-Low-Variance Model via Regularization (50 points) . . . . .	240
A.2.1	a) Why Regularization? . . . . .	240
A.2.2	b) Which Degrees to Exclude? . . . . .	240
A.2.3	c) Apply Ridge Regularization . . . . .	241
A.2.4	d) Visualize Regularized Results . . . . .	241
A.2.5	e) Evaluate the Regularized Model . . . . .	241
A.2.6	f) Interpreting the Impact of Regularization . . . . .	242



# Preface

This book serves as the course notes for STAT 303 Sec20, Spring 2025 at Northwestern University. To enhance your understanding of the material, you are expected to read the textbook before using these notes.

It is an evolving resource designed to support the course's learning objectives. This edition builds upon the foundational work of Professor Arvind Krishna, whose contributions have provided a strong framework for this resource. We are deeply grateful for his efforts, which continue to shape the book's development.

Throughout the quarter, the content will be updated and refined in real time to enhance clarity, depth, and relevance. These modifications ensure alignment with current teaching objectives and methodologies.

As a living document, this book welcomes feedback, suggestions, and contributions from students, instructors, and the broader academic community. Your input helps improve its quality and effectiveness.

Thank you for being part of this journey—we hope this resource serves as a valuable guide in your learning.

## **Part I**

# **Bias & Variance; KNN**



# 1 Bias-variance tradeoff

*Read section 2.2.2 of the book before using these notes.*

*Note that in this course, lecture notes are not sufficient, you must read the book for better understanding. Lecture notes are just implementing the concepts of the book on a dataset, but not explaining the concepts elaborately.*

In this chapter, we will show that a flexible model is likely to have high variance and low bias, while a relatively less flexible model is likely to have a high bias and low variance.

The examples considered below are motivated from the examples shown in the documentation of the `bias_variance_decomp()` function from the `mlxtend` library. We will first manually compute the bias and variance for understanding of the concept. Later, we will show application of the `bias_variance_decomp()` function to estimate bias and variance.

```
# Importing necessary libraries
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error
from sklearn.tree import DecisionTreeRegressor
sns.set(font_scale=1.35)
```

## 1.1 Simple model (Less flexible)

Let us consider a linear regression model as the less-flexible (*or relatively simple*) model.

We will first simulate the test dataset for which we will compute the bias and variance.

```
np.random.seed(101)

# Simulating predictor values of test data
xtest = np.random.uniform(-15, 10, 200)
```

```

# Assuming the true mean response is square of the predictor value
fxtest = xtest**2

# Simulating noiseless test response
ytest = fxtest

# We will find bias and variance using a linear regression model for prediction
model = LinearRegression()

# Visualizing the data and the true mean response
sns.scatterplot(x = xtest, y = ytest)
sns.lineplot(x = xtest, y = fxtest, color = 'grey', linewidth = 2)

# Initializing objects to store predictions and mean squared error
# of 100 models developed on 100 distinct training datasets samples
pred_test = []; mse_test = []

# Iterating over each of the 100 models
for i in range(100):
    np.random.seed(i)

    # Simulating the ith training data
    x = np.random.uniform(-15, 10, 200)
    fx = x**2
    y = fx + np.random.normal(0, 10, 200)

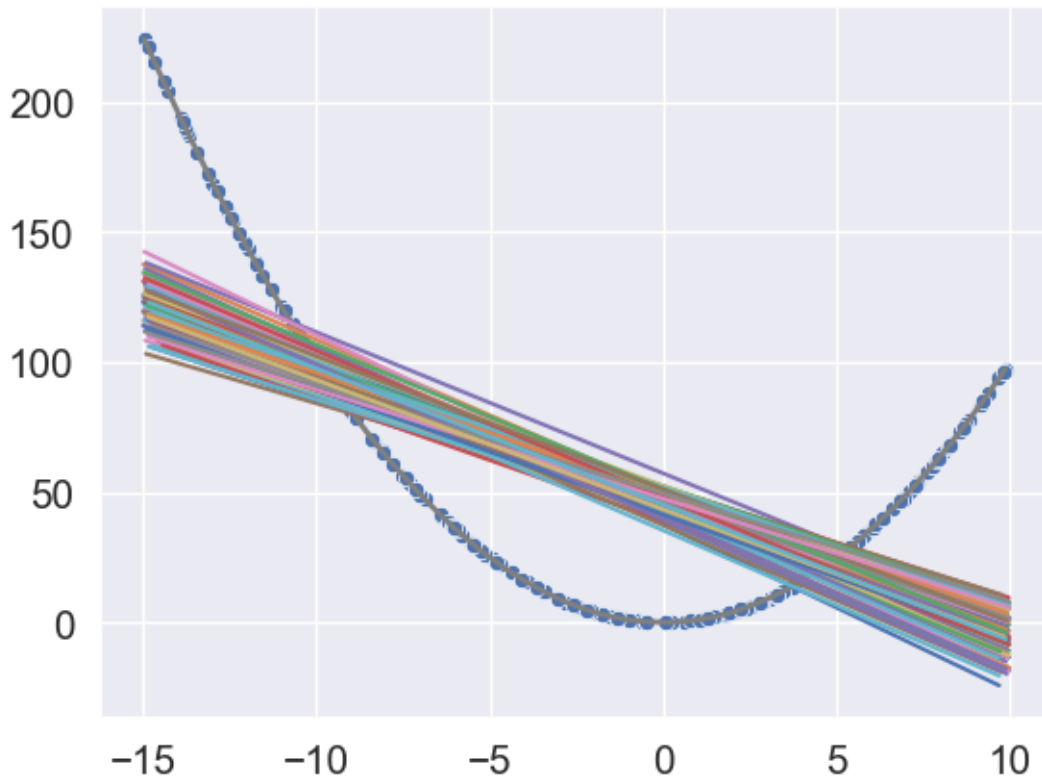
    # Fitting the ith model on the ith training data
    model.fit(x.reshape(-1,1), y)

    # Plotting the ith model
    sns.lineplot(x = x, y = model.predict(x.reshape(-1,1)))

    # Storing the predictions of the ith model on test data
    pred_test.append(model.predict(xtest.reshape(-1,1)))

    # Storing the mean squared error of the ith model on test data
    mse_test.append(mean_squared_error(model.predict(xtest.reshape(-1,1)), ytest))

```



The above plots show that the 100 models seem to have low variance, but high bias. Note that the bias is low only around a couple of points ( $x = -10$  &  $x = 5$ ).

Let us compute the average squared bias over all the test data points.

```
mean_pred = np.array(pred_test).mean(axis = 0)
sq_bias = ((mean_pred - fxtest)**2).mean()
sq_bias
```

2042.104126728109

Let us compute the average variance over all the test data points.

```
mean_var = np.array(pred_test).var(axis = 0).mean()
mean_var
```

28.37397844429763

Let us compute the mean squared error over all the test data points.

```
np.array(mse_test).mean()
```

2070.4781051724062

Note that the mean squared error should be the same as the sum of squared bias, variance, and irreducible error.

The sum of squared bias, model variance, and irreducible error is:

```
sq_bias + mean_var + 100
```

2170.4781051724067

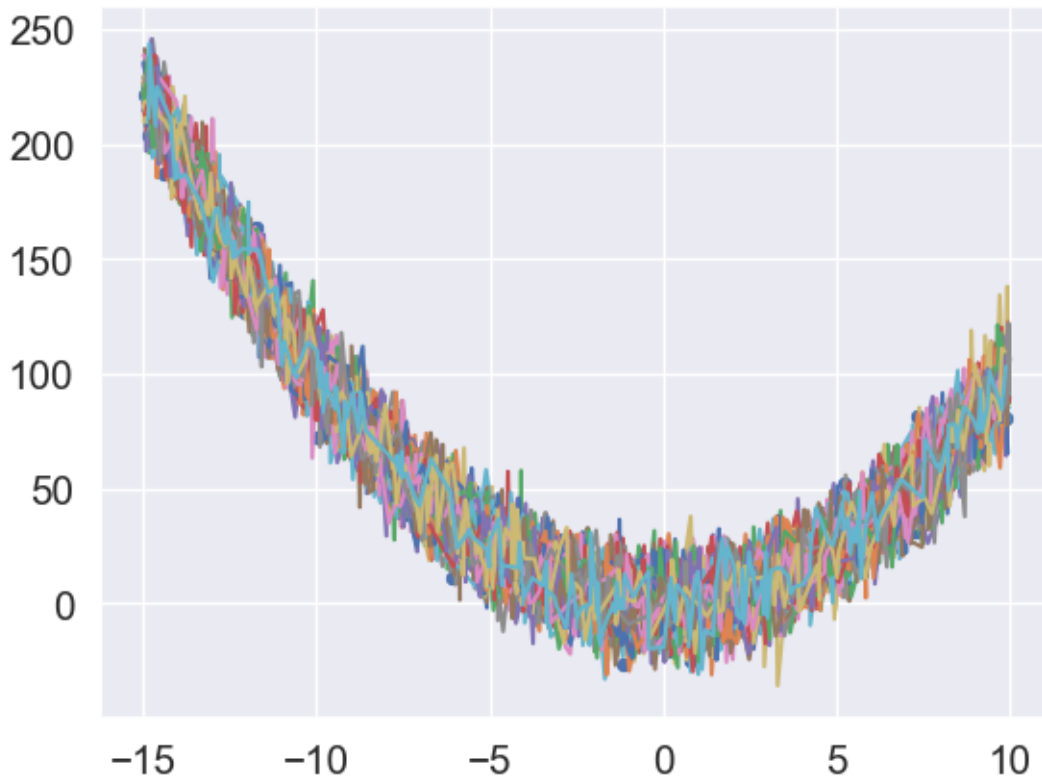
Note that this is approximately, but not exactly, the same as the mean squared error computed above as we are developing a finite number of models, and making predictions on a finite number of test data points.

## 1.2 Complex model (more flexible)

Let us consider a decision tree as the more flexible model.

```
np.random.seed(101)
xtest = np.random.uniform(-15, 10, 200)
fxtest = xtest**2
ytest = fxtest + np.random.normal(0, 10, 200)
model = DecisionTreeRegressor()
```

```
sns.scatterplot(x = xtest, y = ytest)
sns.lineplot(x = xtest, y = fxtest, color = 'grey', linewidth = 2)
pred_test = []; mse_test = []
for i in range(100):
    np.random.seed(i)
    x = np.random.uniform(-15, 10, 200)
    fx = x**2
    y = fx + np.random.normal(0, 10, 200)
    model.fit(x.reshape(-1,1), y)
    sns.lineplot(x = x, y = model.predict(x.reshape(-1,1)))
    pred_test.append(model.predict(xtest.reshape(-1,1)))
    mse_test.append(mean_squared_error(model.predict(xtest.reshape(-1,1)), ytest))
```



The above plots show that the 100 models seem to have high variance, but low bias.

Let us compute the average squared bias over all the test data points.

```
mean_pred = np.array(pred_test).mean(axis = 0)
sq_bias = ((mean_pred - fxtest)**2).mean()
sq_bias
```

```
1.3117561629333938
```

Let us compute the average model variance over all the test data points.

```
mean_var = np.array(pred_test).var(axis = 0).mean()
mean_var
```

```
102.5226748977198
```

Let us compute the average mean squared error over all the test data points.

```
np.array(mse_test).mean()
```

```
225.92027460924726
```

Note that the above error is approximately the same as the sum of the squared bias, model variance and the irreducible error.

Note that the relatively more flexible model has a higher variance, but lower bias as compared to the less flexible linear model. This will typically be the case, but may not be true in all scenarios. We will discuss one such scenario later.

## 2 KNN

*Read section 4.7.6 of the book before using these notes.*

*Note that in this course, lecture notes are not sufficient, you must read the book for better understanding. Lecture notes are just implementing the concepts of the book on a dataset, but not explaining the concepts elaborately.*

```
# Importing necessary libraries
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
sns.set(font_scale=1.35)

from sklearn.preprocessing import StandardScaler
from sklearn.neighbors import KNeighborsRegressor, KNeighborsClassifier
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import cross_val_score, GridSearchCV, cross_val_predict, KFold,
```

### 2.1 KNN for regression

```
#Using the same datasets as used for linear regression in STAT303-2,
#so that we can compare the non-linear models with linear regression
trainf = pd.read_csv('./Datasets/Car_features_train.csv')
trainp = pd.read_csv('./Datasets/Car_prices_train.csv')
testf = pd.read_csv('./Datasets/Car_features_test.csv')
testp = pd.read_csv('./Datasets/Car_prices_test.csv')
train = pd.merge(trainf, trainp)
test = pd.merge(testf, testp)
train.head()
```

	carID	brand	model	year	transmission	mileage	fuelType	tax	mpg	engineSize	price
0	18473	bmw	6 Series	2020	Semi-Auto	11	Diesel	145	53.3282	3.0	37980
1	15064	bmw	6 Series	2019	Semi-Auto	10813	Diesel	145	53.0430	3.0	33980
2	18268	bmw	6 Series	2020	Semi-Auto	6	Diesel	145	53.4379	3.0	36850
3	18480	bmw	6 Series	2017	Semi-Auto	18895	Diesel	145	51.5140	3.0	25998
4	18492	bmw	6 Series	2015	Automatic	62953	Diesel	160	51.4903	3.0	18990

```

predictors = ['mpg', 'engineSize', 'year', 'mileage']

X_train = train[predictors]
y_train = train['price']

X_test = test[predictors]
y_test = test['price']

```

Let us scale data as we are using KNN.

### 2.1.1 Scaling data

```

# Scale
sc = StandardScaler()

sc.fit(X_train)
X_train_scaled = sc.transform(X_train)
X_test_scaled = sc.transform(X_test)

```

Let fit the model and compute the RMSE on test data. If the number of neighbors is not specified, the default value is taken.

### 2.1.2 Fitting and validating model

```

knn_model = KNeighborsRegressor()

knn_model.fit(X_train_scaled, (y_train))

y_pred = knn_model.predict(X_test_scaled)
y_pred_train = knn_model.predict(X_train_scaled)

```



```
mean_squared_error(y_test, (y_pred), squared=False)
```

```
6329.691192885354
```

```
knn_model2 = KNeighborsRegressor(n_neighbors = 5, weights='distance') # Default weights is uniform
knn_model2.fit(X_train_scaled, y_train)
y_pred = knn_model2.predict(X_test_scaled)
mean_squared_error(y_test, y_pred, squared=False)
```

```
6063.327598353961
```

The model seems to fit better than all the linear models in STAT303-2.

### 2.1.3 Hyperparameter tuning

We will use cross-validation to find the optimal value of the hyperparameter `n_neighbors`.

```
Ks = np.arange(1,601)

cv_scores = []

for K in Ks:
    model = KNeighborsRegressor(n_neighbors = K, weights='distance')
    score = cross_val_score(model, X_train_scaled, y_train, cv=5, scoring = 'neg_root_mean_s
    cv_scores.append(score)
```

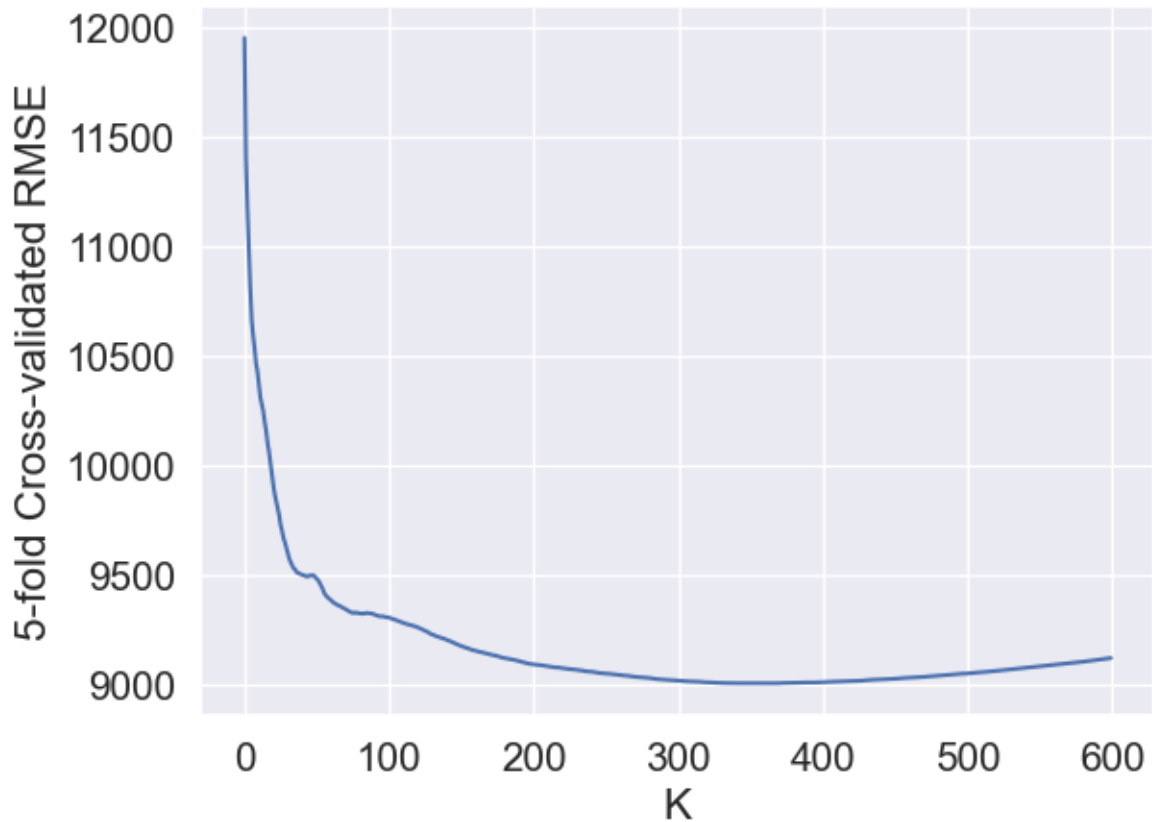
```
np.array(cv_scores).shape
# Each row is a K
```

```
(600, 5)
```

```
cv_scores_array = np.array(cv_scores)

avg_cv_scores = -cv_scores_array.mean(axis=1)
```

```
sns.lineplot(x = range(600), y = avg_cv_scores);  
plt.xlabel('K')  
plt.ylabel('5-fold Cross-validated RMSE');
```



```
avg_cv_scores.min() # Best CV score  
Ks[avg_cv_scores.argmin()] # Best hyperparam value
```

366

The optimal hyperparameter value is 366. Does it seem to be too high?

```
best_model = KNeighborsRegressor(n_neighbors = Ks[avg_cv_scores.argmin()], weights='distance'  
best_model.fit(X_train_scaled, y_train)
```

```
y_pred = best_model.predict(X_test_scaled)

mean_squared_error(y_test, y_pred, squared=False)
```

7724.452068618346

**The test error with the optimal hyperparameter value based on cross-validation is much higher than that based on the default value of the hyperparameter.** Why is that?

Sometimes this may happen by chance due to the specific observations in the  $k$  folds. One option is to shuffle the dataset before splitting into folds.

The function `KFold()` can be used to shuffle the data before splitting it into folds.

### 2.1.3.1 `KFold()`

```
kcv = KFold(n_splits = 5, shuffle = True, random_state = 1)
```

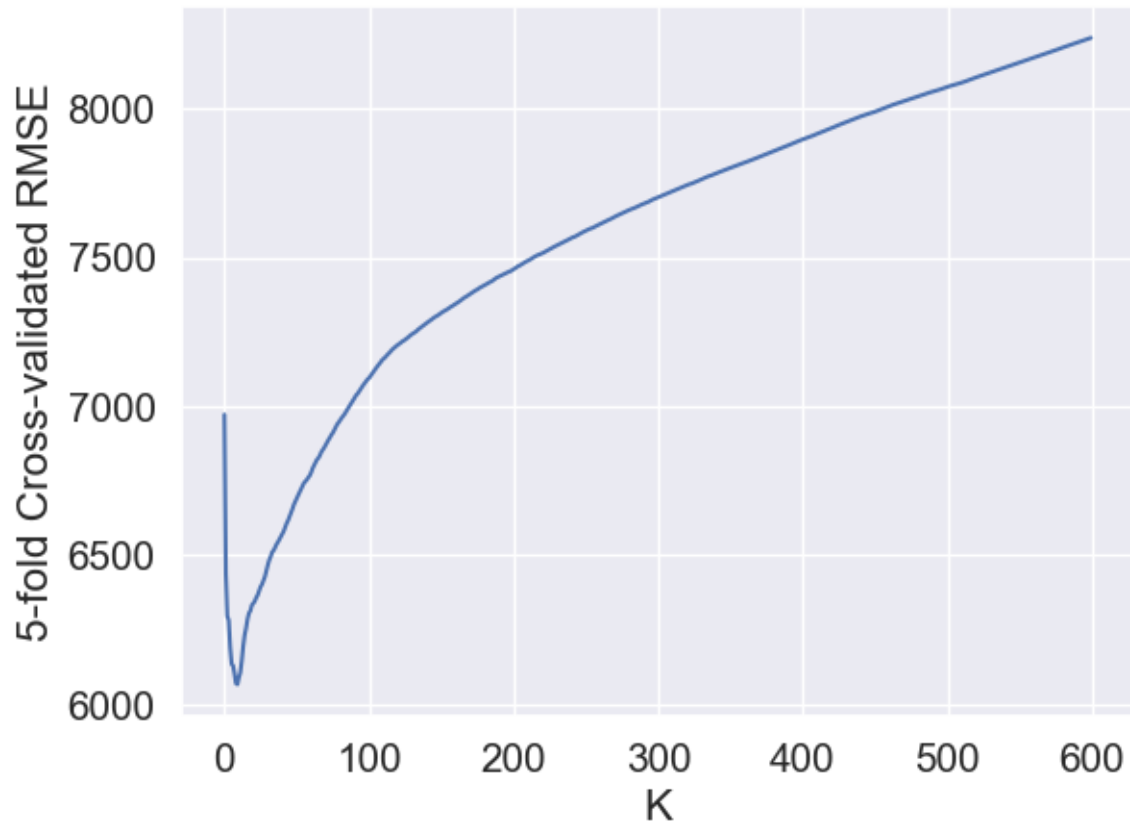
Now, let us again try to find the optimal  $K$  for KNN, using the new folds, based on shuffled data.

```
Ks = np.arange(1,601)

cv_scores = []

for K in Ks:
    model = KNeighborsRegressor(n_neighbors = K, weights='distance')
    score = cross_val_score(model, X_train_scaled, y_train, cv = kcv, scoring = 'neg_root_mean_squared_error')
    cv_scores.append(score)
```

```
cv_scores_array = np.array(cv_scores)
avg_cv_scores = -cv_scores_array.mean(axis=1)
sns.lineplot(x = range(600), y = avg_cv_scores);
plt.xlabel('K')
plt.ylabel('5-fold Cross-validated RMSE');
```



The optimal K is:

```
Ks[avg_cv_scores.argmin()]
```

10

RMSE on test data with this optimal value of  $K$  is:

```
knn_model2 = KNeighborsRegressor(n_neighbors = 10, weights='distance') # Default weights is v
knn_model2.fit(X_train_scaled, y_train)
y_pred = knn_model2.predict(X_test_scaled)
mean_squared_error(y_test, y_pred, squared=False)
```

6043.889393238132

In order to avoid these errors due the specific observations in the  $k$  folds, it will be better to repeat the  $k$ -fold cross-validation multiple times, where the data is shuffled after each  $k$ -fold cross-validation, so that the cross-validation takes place on new folds for each repetition.

The function `RepeatedKfold()` repeats  $k$ -fold cross validation multiple times (*10 times by default*). Let us use it to have a more robust optimal value of the number of neighbors  $K$ .

### 2.1.3.2 `RepeatedKfold()`

```
kcv = RepeatedKfold(n_splits = 5, random_state = 1)
```

```
Ks = np.arange(1,601)
```

```
cv_scores = []
```

```
for K in Ks:
```

```
    model = KNeighborsRegressor(n_neighbors = K, weights='distance')
```

```
    score = cross_val_score(model, X_train_scaled, y_train, cv = kcv, scoring = 'neg_root_me
```

```
    cv_scores.append(score)
```

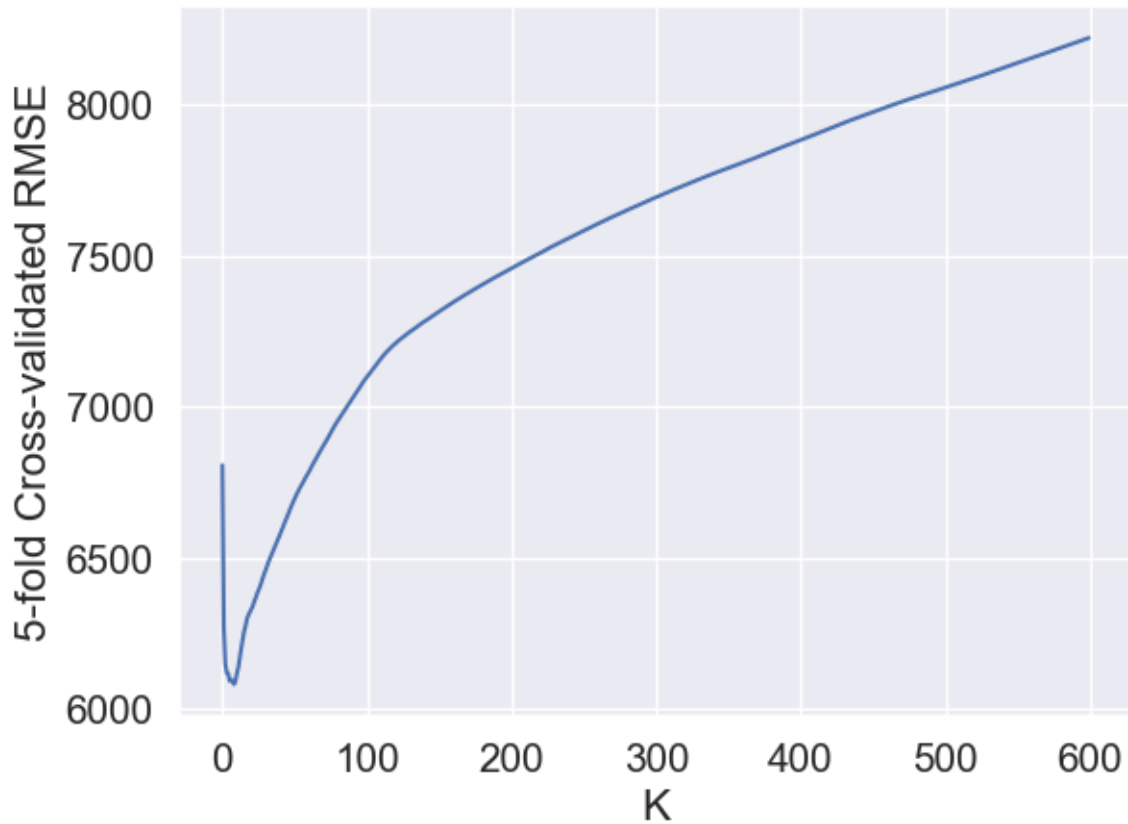
```
cv_scores_array = np.array(cv_scores)
```

```
avg_cv_scores = -cv_scores_array.mean(axis=1)
```

```
sns.lineplot(x = range(600), y = avg_cv_scores);
```

```
plt.xlabel('K')
```

```
plt.ylabel('5-fold Cross-validated RMSE');
```



The optimal K is:

```
Ks[avg_cv_scores.argmin()]
```

9

RMSE on test data with this optimal value of  $K$  is:

```
knn_model2 = KNeighborsRegressor(n_neighbors = 9, weights='distance') # Default weights is u  
knn_model2.fit(X_train_scaled, y_train)  
y_pred = knn_model2.predict(X_test_scaled)  
mean_squared_error(y_test, y_pred, squared=False)
```

6051.157910333279

### 2.1.4 KNN hyperparameters

The model hyperparameters can be obtained using the `get_params()` method. Note that there are other hyperparameters to tune in addition to number of neighbors. However, the number of neighbours may be the most influential hyperparameter in most cases.

```
best_model.get_params()
```

```
{'algorithm': 'auto',  
 'leaf_size': 30,  
 'metric': 'minkowski',  
 'metric_params': None,  
 'n_jobs': None,  
 'n_neighbors': 366,  
 'p': 2,  
 'weights': 'distance'}
```

The distances and the indices of the nearest K observations to each test observation can be obtained using the `kneighbors()` method.

```
best_model.kneighbors(X_test_scaled, return_distance=True)
```

```
# Each row is a test obs
```

```
# The cols are the indices of the K Nearest Neighbors (in the training data) to the test obs
```

```
(array([[1.92799060e-02, 1.31899013e-01, 1.89662146e-01, ...,  
        8.38960707e-01, 8.39293053e-01, 8.39947823e-01],  
       [7.07215830e-02, 1.99916181e-01, 2.85592939e-01, ...,  
        1.15445056e+00, 1.15450848e+00, 1.15512897e+00],  
       [1.32608205e-03, 1.43558347e-02, 1.80622215e-02, ...,  
        5.16758453e-01, 5.17378567e-01, 5.17852312e-01],  
       ...,  
       [1.29209535e-02, 1.59187173e-02, 3.67038947e-02, ...,  
        8.48811744e-01, 8.51235616e-01, 8.55044146e-01],  
       [1.84971803e-02, 1.67471541e-01, 1.69374312e-01, ...,  
        7.76743422e-01, 7.76943691e-01, 7.77760930e-01],  
       [4.63762129e-01, 5.88639393e-01, 7.54718535e-01, ...,  
        3.16994824e+00, 3.17126663e+00, 3.17294300e+00]]),  
 array([[1639, 1647, 4119, ..., 3175, 2818, 4638],  
       [ 367, 1655, 1638, ..., 2010, 3600,  268],  
       [ 393, 4679, 3176, ..., 4663,  357,  293],
```

```
...,
[3116, 3736, 3108, ..., 3841, 2668, 2666],
[4864, 3540, 4852, ..., 3596, 3605, 4271],
[ 435,  729, 4897, ..., 4112, 2401, 2460]], dtype=int64))
```

## 2.2 KNN for classification

KNN model for classification can be developed and tuned in a similar manner using the sklearn function `KNeighborsClassifier()`

- For classification, `KNeighborsClassifier`
- Exact same inputs
  - One detail: Not common to use even numbers for K in classification because of majority voting
  - `Ks = np.arange(1,41,2)` -> To get the odd numbers



## 3 Hyperparameter tuning

In this chapter we'll introduce several functions that help with tuning hyperparameters of a machine learning model.

```
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split, cross_val_score, cross_val_predict, \
cross_validate, GridSearchCV, RandomizedSearchCV, KFold, StratifiedKFold, RepeatedKFold, Rep
from sklearn.neighbors import KNeighborsClassifier, KNeighborsRegressor
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import accuracy_score, recall_score, mean_squared_error
from scipy.stats import uniform
from skopt import BayesSearchCV
from skopt.space import Real, Categorical, Integer
import seaborn as sns
from skopt.plots import plot_objective, plot_histogram, plot_convergence
import matplotlib.pyplot as plt
import warnings
from IPython import display
```

Let us read and pre-process data first. Then we'll be ready to tune the model hyperparameters. We'll use KNN as the model. Note that KNN has multiple hyperparameters to tune, such as number of neighbors, distance metric, weights of neighbours, etc.

```
#Using the same datasets as used for linear regression in STAT303-2,
#so that we can compare the non-linear models with linear regression
trainf = pd.read_csv('./Datasets/Car_features_train.csv')
trainp = pd.read_csv('./Datasets/Car_prices_train.csv')
testf = pd.read_csv('./Datasets/Car_features_test.csv')
testp = pd.read_csv('./Datasets/Car_prices_test.csv')
train = pd.merge(trainf, trainp)
test = pd.merge(testf, testp)
train.head()
```

	carID	brand	model	year	transmission	mileage	fuelType	tax	mpg	engineSize	price
0	18473	bmw	6 Series	2020	Semi-Auto	11	Diesel	145	53.3282	3.0	37980
1	15064	bmw	6 Series	2019	Semi-Auto	10813	Diesel	145	53.0430	3.0	33980
2	18268	bmw	6 Series	2020	Semi-Auto	6	Diesel	145	53.4379	3.0	36850
3	18480	bmw	6 Series	2017	Semi-Auto	18895	Diesel	145	51.5140	3.0	25998
4	18492	bmw	6 Series	2015	Automatic	62953	Diesel	160	51.4903	3.0	18990

```

predictors = ['mpg', 'engineSize', 'year', 'mileage']
X_train = train[predictors]
y_train = train['price']
X_test = test[predictors]
y_test = test['price']

# Scale
sc = StandardScaler()

sc.fit(X_train)
X_train_scaled = sc.transform(X_train)
X_test_scaled = sc.transform(X_test)

```

### 3.1 GridSearchCV

The function is used to compute the cross-validated score (*MSE*, *RMSE*, *accuracy*, *etc.*) over a grid of hyperparameter values. This helps avoid nested `for()` loops if multiple hyperparameter values need to be tuned.

```

# GridSearchCV works in three steps:

# 1) Create the model
model = KNeighborsRegressor() # No inputs defined inside the model

# 2) Create a hyperparameter grid (as a dict)
# the keys should be EXACTLY the same as the names of the model inputs
# the values should be an array or list of hyperparam values you want to try out

# 30 K values x 2 weight settings x 3 metric settings = 180 different combinations in this grid
grid = {'n_neighbors': np.arange(5, 151, 5), 'weights':['uniform', 'distance'],
        'metric': ['manhattan', 'euclidean', 'chebyshev']}

# 3) Create the Kfold object (Using RepeatedKfold will be more robust, but more expensive, use
# have the budget)

```

```

kfold = KFold(n_splits = 5, shuffle = True, random_state = 1)

# 4) Create the CV object
# Look at the documentation to see the order in which the objects must be specified within the GridSearchCV
gcv = GridSearchCV(model, grid, cv = kfold, scoring = 'neg_root_mean_squared_error', n_jobs=-1)

# Fit the models, and cross-validate
gcv.fit(X_train_scaled, y_train)

```

Fitting 5 folds for each of 180 candidates, totalling 900 fits

```

GridSearchCV(cv=KFold(n_splits=5, random_state=1, shuffle=True),
             estimator=KNeighborsRegressor(), n_jobs=-1,
             param_grid={'metric': ['manhattan', 'euclidean', 'chebyshev'],
                         'n_neighbors': array([ 5, 10, 15, 20, 25, 30, 35, 40, 45,
70, 75, 80, 85, 90, 95, 100, 105, 110, 115, 120, 125, 130,
135, 140, 145, 150]),
                         'weights': ['uniform', 'distance']}},
             scoring='neg_root_mean_squared_error', verbose=10)

```

The optimal estimator based on cross-validation is:

```
gcv.best_estimator_
```

```
KNeighborsRegressor(metric='manhattan', n_neighbors=10, weights='distance')
```

The optimal hyperparameter values (*based on those considered in the grid search*) are:

```
gcv.best_params_
```

```
{'metric': 'manhattan', 'n_neighbors': 10, 'weights': 'distance'}
```

The cross-validated root mean squared error for the optimal hyperparameter values is:

```
-gcv.best_score_
```

```
5740.928686723918
```

The RMSE on test data for the optimal hyperparameter values is:

```
y_pred = gcv.predict(X_test_scaled)
mean_squared_error(y_test, y_pred, squared=False)
```

5747.466851437544

Note that the error is further reduced as compared to the case when we tuned only one hyperparameter in the [previous chapter](#). We must tune all the hyperparameters that can effect prediction accuracy, in order to get the most accurate model.

The results for each cross-validation are stored in the `cv_results_` attribute.

```
pd.DataFrame(gcv.cv_results_).head()
```

	mean_fit_time	std_fit_time	mean_score_time	std_score_time	param_metric	param_n_neighb
0	0.011169	0.005060	0.011768	0.001716	manhattan	5
1	0.009175	0.001934	0.009973	0.000631	manhattan	5
2	0.008976	0.001092	0.012168	0.001323	manhattan	10
3	0.007979	0.000001	0.011970	0.000892	manhattan	10
4	0.006781	0.000748	0.012367	0.001017	manhattan	15

These results can be useful to see if other hyperparameter values are almost equally good.

For example, the next two best optimal values of the hyperparameter correspond to neighbors being 15 and 5 respectively. As the test error has a high variance, the best hyperparameter values need not necessarily be actually optimal.

```
pd.DataFrame(gcv.cv_results_).sort_values(by = 'rank_test_score').head()
```

	mean_fit_time	std_fit_time	mean_score_time	std_score_time	param_metric	param_n_neighb
3	0.007979	0.000001	0.011970	0.000892	manhattan	10
5	0.009374	0.004829	0.013564	0.001850	manhattan	15
1	0.009175	0.001934	0.009973	0.000631	manhattan	5
7	0.007977	0.001092	0.017553	0.002054	manhattan	20
9	0.007777	0.000748	0.019349	0.003374	manhattan	25

Let us compute the RMSE on test data based on the 2nd and 3rd best hyperparameter values.

```
model = KNeighborsRegressor(n_neighbors=15, metric='manhattan', weights='distance').fit(X_train_scaled, y_train_scaled)
mean_squared_error(model.predict(X_test_scaled), y_test, squared = False)
```

5800.418957612656

```
model = KNeighborsRegressor(n_neighbors=5, metric='manhattan', weights='distance').fit(X_train_scaled, y_train_scaled)
mean_squared_error(model.predict(X_test_scaled), y_test, squared = False)
```

5722.4859230146685

We can see that the RMSE corresponding to the 3rd best hyperparameter value is the least. Due to variance in test errors, it may be a good idea to consider the set of top few best hyperparameter values, instead of just considering the best one.

## 3.2 RandomizedSearchCV()

In case of many possible values of hyperparameters, it may be computationally very expensive to use GridSearchCV(). In such cases, RandomizedSearchCV() can be used to compute the cross-validated score on a randomly selected subset of hyperparameter values from the specified grid. The number of values can be fixed by the user, as per the available budget.

```
# RandomizedSearchCV works in three steps:

# 1) Create the model
model = KNeighborsRegressor() # No inputs defined inside the model

# 2) Create a hyperparameter grid (as a dict)
# the keys should be EXACTLY the same as the names of the model inputs
# the values should be an array or list of hyperparam values, or distribution of hyperparameters

grid = {'n_neighbors': range(1, 500), 'weights': ['uniform', 'distance'],
        'metric': ['minkowski'], 'p': uniform(loc=1, scale=10)} #We can specify a distribution
                                                                #for continuous hyperparameters

# 3) Create the Kfold object (Using RepeatedKfold will be more robust, but more expensive, use it if budget allows)
# have the budget)
kfold = KFold(n_splits = 5, shuffle = True, random_state = 1)
```

```
# 4) Create the CV object
# Look at the documentation to see the order in which the objects must be specified within the
gcv = RandomizedSearchCV(model, param_distributions = grid, cv = kfold, n_iter = 180, random_state=10,
                        scoring = 'neg_root_mean_squared_error', n_jobs = -1, verbose = 10)

# Fit the models, and cross-validate
gcv.fit(X_train_scaled, y_train)
```

Fitting 5 folds for each of 180 candidates, totalling 900 fits

```
RandomizedSearchCV(cv=KFold(n_splits=5, random_state=1, shuffle=True),
                  estimator=KNeighborsRegressor(), n_iter=180, n_jobs=-1,
                  param_distributions={'metric': ['minkowski'],
                                       'n_neighbors': range(1, 500),
                                       'p': <scipy.stats._distn_infrastructure.rv_continuous object>,
                                       'weights': ['uniform', 'distance']},
                  random_state=10, scoring='neg_root_mean_squared_error',
                  verbose=10)
```

```
gcv.best_params_
```

```
{'metric': 'minkowski',
 'n_neighbors': 3,
 'p': 1.252639454318171,
 'weights': 'uniform'}
```

```
gcv.best_score_
```

```
-6239.171627183809
```

```
y_pred = gcv.predict(X_test_scaled)
mean_squared_error(y_test, y_pred, squared=False)
```

```
6176.533397589911
```

Note that in this example, `RandomizedSearchCV()` helps search for optimal values of the hyperparameter  $p$  over a continuous domain space. In this dataset,  $p = 1$  seems to be the optimal value. However, if the optimal value was somewhere in the middle of a larger

continuous domain space (*instead of the boundary of the domain space*), and there were several other hyperparameters, some of which were not influencing the response (*effect sparsity*), `RandomizedSearchCV()` is likely to be more effective in estimating the optimal value of the continuous hyperparameter.

The advantages of `RandomizedSearchCV()` over `GridSearchCV()` are:

1. `RandomizedSearchCV()` fixes the computational cost in case of large number of hyperparameters / large number of levels of individual hyperparameters. If there are  $n$  hyperparameters, each with 3 levels, the number of all possible hyperparameter values will be  $3^n$ . The computational cost increase exponentially with increase in number of hyperparameters.
2. In case of a hyperparameter having continuous values, the distribution of the hyperparameter can be specified in `RandomizedSearchCV()`.
3. In case of effect sparsity of hyperparameters, i.e., if only a few hyperparameters significantly effect prediction accuracy, `RandomizedSearchCV()` is likely to consider more unique values of the influential hyperparameters as compared to `GridSearchCV()`, and is thus likely to provide more optimal hyperparameter values as compared to `GridSearchCV()`. The figure below shows effect sparsity where there are 2 hyperparameters, but only one of them is associated with the cross-validated score, Here, it is more likely that the optimal cross-validated score will be obtained by `RandomizedSearchCV()`, as it is evaluating the model on 9 unique values of the relevant hyperparameter, instead of just 3.

<IPython.core.display.Image object>

### 3.3 `BayesSearchCV()`

Unlike the grid search and random search, which treat hyperparameter sets independently, the Bayesian optimization is an informed search method, meaning that it learns from previous iterations. The number of trials in this approach is determined by the user.

- The function begins by computing the cross-validated score by randomly selecting a few hyperparameter values from the specified distribution of hyperparameter values.
- Based on the data of hyperparameter values tested (*predictors*), and the cross-validated score (*the response*), a Gaussian process model is developed to estimate the cross-validated score & the uncertainty in the estimate in the entire space of the hyperparameter values

- A criterion that “explores” uncertain regions of the space of hyperparameter values (*where it is difficult to predict cross-validated score*), and “exploits” promising regions of the space of hyperparameter values (*where the cross-validated score is predicted to minimize*) is used to suggest the next hyperparameter value that will potentially minimize the cross-validated score
- Cross-validated score is computed at the suggested hyperparameter value, the Gaussian process model is updated, and the previous step is repeated, until a certain number of iterations specified by the user.

To summarize, instead of blindly testing the model for the specified hyperparameter values (*as in `GridSearchCV()`*), or randomly testing the model on certain hyperparameter values (*as in `RandomizedSearchCV()`*), `BayesSearchCV()` smartly tests the model for those hyperparameter values that are likely to reduce the cross-validated score. The algorithm becomes “smarter” as it “learns” more with increasing iterations.

Here is a nice [blog](#), if you wish to understand more about the Bayesian optimization procedure.

```
# BayesSearchCV works in three steps:

# 1) Create the model
model = KNeighborsRegressor(metric = 'minkowski') # No inputs defined inside the model

# 2) Create a hyperparameter grid (as a dict)
# the keys should be EXACTLY the same as the names of the model inputs
# the values should be the distribution of hyperparameter values. Lists and NumPy arrays can
# also be used

grid = {'n_neighbors': Integer(1, 500), 'weights': Categorical(['uniform', 'distance']),
        'p': Real(1, 10, prior = 'uniform')}

# 3) Create the Kfold object (Using RepeatedKfold will be more robust, but more expensive,
# use it if you have the budget)
kfold = KFold(n_splits = 5, shuffle = True, random_state = 1)

# 4) Create the CV object
# Look at the documentation to see the order in which the objects must be specified within
# the function
gcv = BayesSearchCV(model, search_spaces = grid, cv = kfold, n_iter = 180, random_state = 10,
                    scoring = 'neg_root_mean_squared_error', n_jobs = -1)

# Fit the models, and cross-validate
```



```
# Sometimes the Gaussian process model predicting the cross-validated score suggests a
# "promising point" (i.e., set of hyperparameter values) for cross-validation that it has
# already suggested earlier. In such a case a warning is raised, and the objective
# function (i.e., the cross-validation score) is computed at a randomly selected point
# (as in RandomizedSearchCV()). This feature helps the algorithm explore other regions of
# the hyperparameter space, rather than only searching in the promising regions. Thus, it
# balances exploration (of the hyperparameter space) with exploitation (of the promising
# regions of the hyperparameter space)

warnings.filterwarnings("ignore")
gcv.fit(X_train_scaled, y_train)
warnings.resetwarnings()
```

The optimal hyperparameter values (*based on Bayesian search*) on the provided distribution of hyperparameter values are:

```
gcv.best_params_
```

```
OrderedDict([('n_neighbors', 9),
             ('p', 1.0008321732366932),
             ('weights', 'distance')])
```

The cross-validated root mean squared error for the optimal hyperparameter values is:

```
-gcv.best_score_
```

```
5756.172382596493
```

The RMSE on test data for the optimal hyperparameter values is:

```
y_pred = gcv.predict(X_test_scaled)
mean_squared_error(y_test, y_pred, squared=False)
```

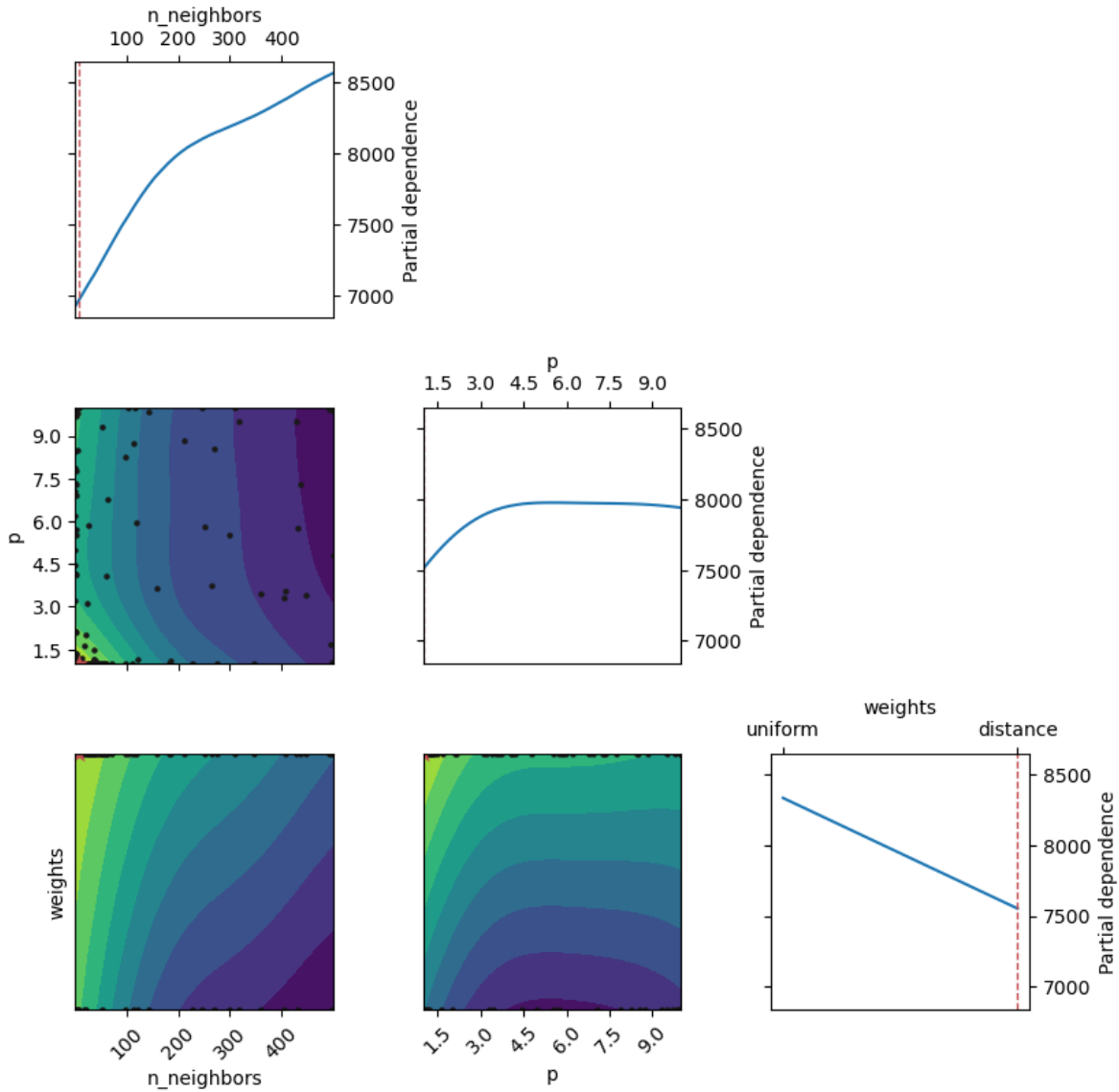
```
5740.432278861367
```

### 3.3.1 Diagnosis of cross-validated score optimization

Below are the partial dependence plots of the objective function (*i.e.*, the cross-validated score). The cross-validated score predictions are based on the most recently updated model (*i.e.*, the updated *Gaussian Process* model at the end of `n_iter` iterations specified by the user) that predicts the cross-validated score.

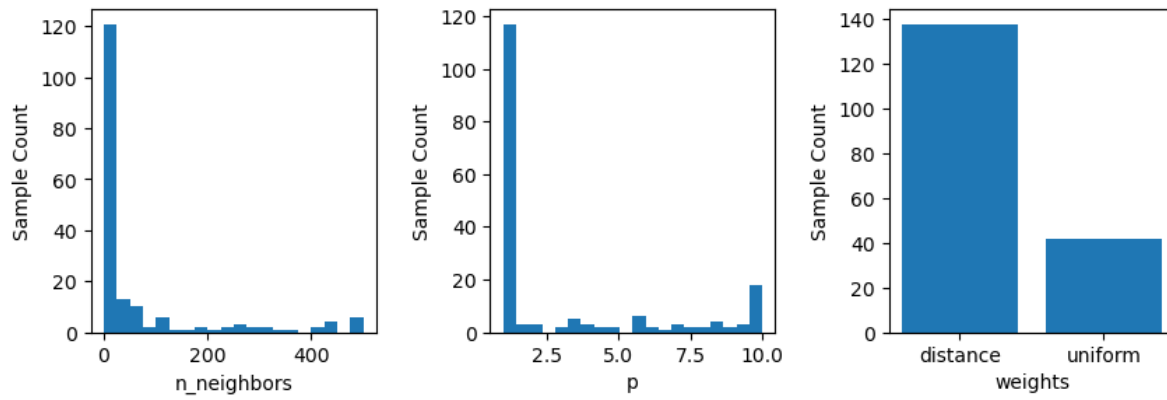
Check the `plot_objective()` documentation to interpret the plots.

```
plot_objective(gcv.optimizer_results_[0],  
               dimensions=["n_neighbors", "p", "weights"], size = 3)  
plt.show();
```



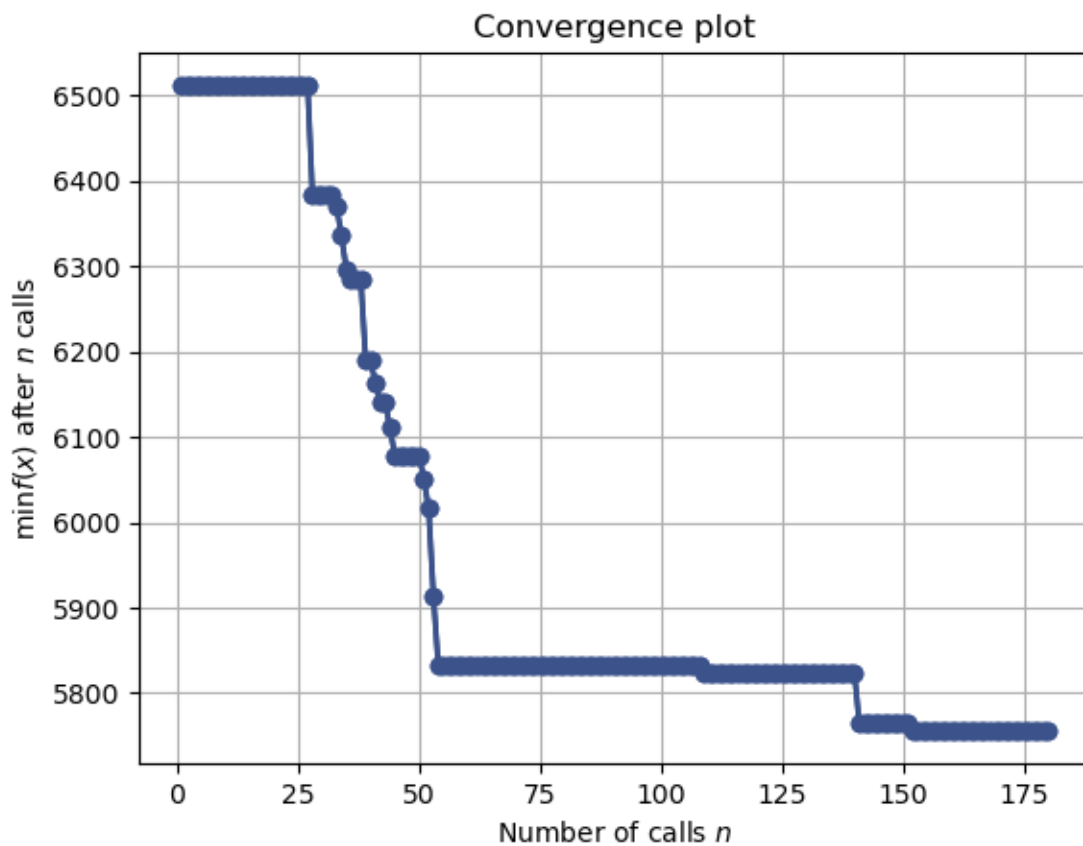
The frequency of individual hyperparameter values considered can also be visualized as below.

```
fig, ax = plt.subplots(1, 3, figsize = (10, 3))
plt.subplots_adjust(wspace=0.4)
plot_histogram(gcv.optimizer_results_[0], 0, ax = ax[0])
plot_histogram(gcv.optimizer_results_[0], 1, ax = ax[1])
plot_histogram(gcv.optimizer_results_[0], 2, ax = ax[2])
plt.show()
```



Below is the plot showing the minimum cross-validated score computed obtained until 'n' hyperparameter values are considered for cross-validation.

```
plot_convergence(gcv.optimizer_results_)
plt.show()
```



Note that the cross-validated error is close to the optimal value in the 53rd iteration itself.

The cross-validated error at the 53rd iteration is:

```
gcv.optimizer_results_[0]['func_vals'][53]
```

```
5831.87280274334
```

The hyperparameter values at the 53rd iterations are:

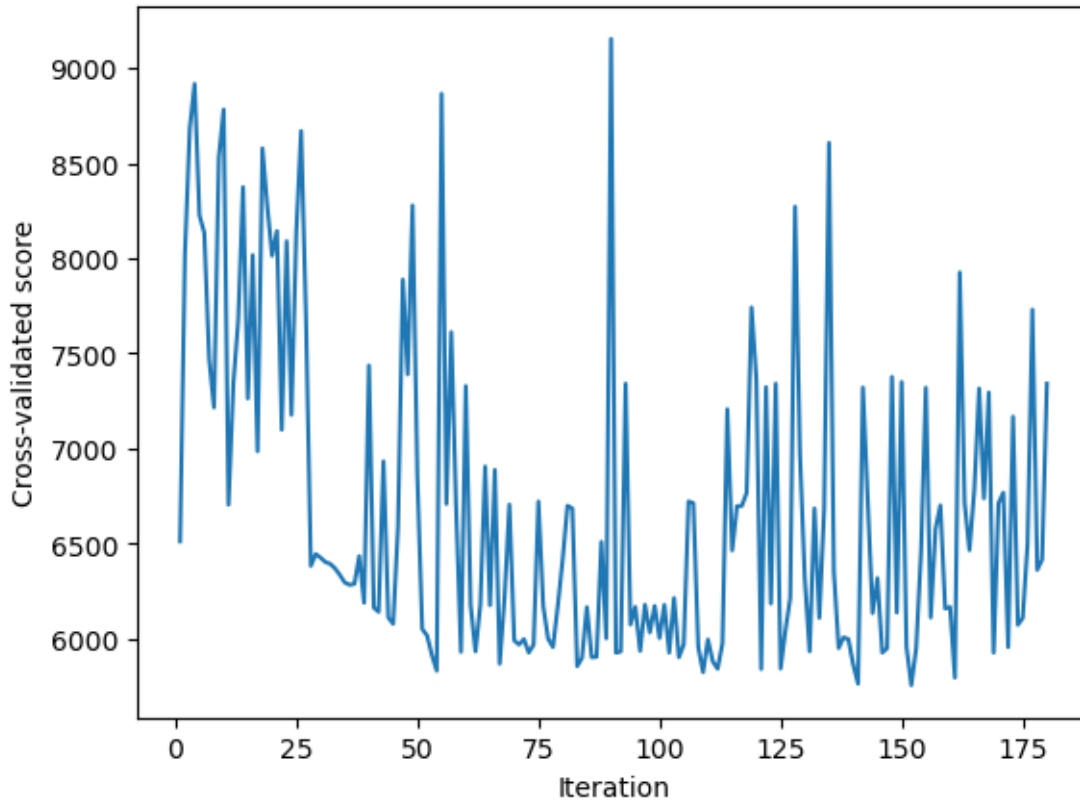
```
gcv.optimizer_results_[0]['x_iters'][53]
```

```
[15, 1.0, 'distance']
```

Note that this is the 2nd most optimal hyperparameter value based on `GridSearchCV()`.

Below is the plot showing the cross-validated score computed at each of the 180 hyperparameter values considered for cross-validation. The plot shows that the algorithm seems to explore new regions of the domain space, instead of just exploiting the promising ones. There is a balance between exploration and exploitation for finding the optimal hyperparameter values that minimize the objective function (*i.e., the function that models the cross-validated score*).

```
sns.lineplot(x = range(1, 181), y = gcv.optimizer_results_[0]['func_vals'])
plt.xlabel('Iteration')
plt.ylabel('Cross-validated score')
plt.show();
```



The advantages of `BayesSearchCV()` over `GridSearchCV()` and `RandomizedSearchCV()` are:

1. The Bayesian Optimization approach gives the benefit that we can give a much larger range of possible values, since over time we identify and exploit the most promising regions and discard the not so promising ones. Plain grid-search would burn computational resources to explore all regions of the domain space with the same granularity, even the not promising ones. Since we search much more effectively in Bayesian search, we can search over a larger domain space.
2. BayesSearch CV may help us identify the optimal hyperparameter value in fewer iterations if the Gaussian process model estimating the cross-validated score is relatively accurate. However, this is not certain. Grid and random search are completely uninformed by past evaluations, and as a result, often spend a significant amount of time evaluating “bad” hyperparameters.
3. BayesSearch CV is more reliable in cases of a large search space, where random selection may miss sampling values from optimal regions of the search space.

The disadvantages of `BayesSearchCV()` over `GridSearchCV()` and `RandomizedSearchCV()` are:

1. `BayesSearchCV()` has a cost of learning from past data, i.e., updating the model that predicts the cross-validated score after every iteration of evaluating the cross-validated score on a new hyperparameter value. This cost will continue to increase as more and more data is collected. There is no such cost in `GridSearchCV()` and `RandomizedSearchCV()` as there is no learning. This implies that each iteration of `BayesSearchCV()` will take a longer time than each iteration of `GridSearchCV()` / `RandomizedSearchCV()`. Thus, even if `BayesSearchCV()` finds the optimal hyperparameter value in fewer iterations, it may take more time than `GridSearchCV()` / `RandomizedSearchCV()` for the same.
2. The success of `BayesSearchCV()` depends on the predictions and associated uncertainty estimated by the Gaussian process (GP) model that predicts the cross-validated score. The GP model, although works well in general, may not be suitable for certain datasets, or may take a relatively large number of iterations to learn for certain datasets.

### 3.3.2 Live monitoring of cross-validated score

Note that it will be useful monitor the cross-validated score while the Bayesian Search CV code is running, and stop the code as soon as the desired accuracy is reached, or the optimal cross-validated score doesn't seem to improve. The `fit()` method of the `BayesSearchCV()` object has a `callback` argument that can be used as follows:

```
model = KNeighborsRegressor(metric = 'minkowski') # No inputs defined inside the model
grid = {'n_neighbors': Integer(1, 500), 'weights': Categorical(['uniform', 'distance']),
        'p': Real(1, 10, prior = 'uniform')}

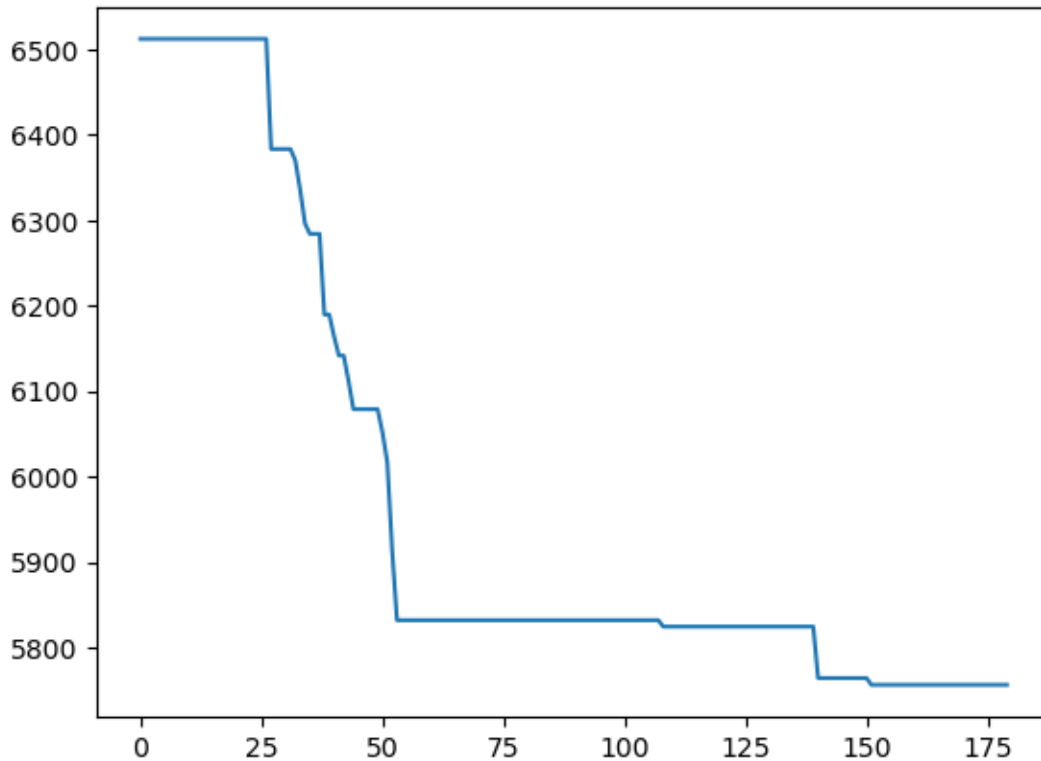
kfold = KFold(n_splits = 5, shuffle = True, random_state = 1)
gcv = BayesSearchCV(model, search_spaces = grid, cv = kfold, n_iter = 180, random_state = 10,
                    scoring = 'neg_root_mean_squared_error', n_jobs = -1)

paras = list(gcv.search_spaces.keys())
paras.sort()

def monitor(optim_result):
    cv_values = pd.Series(optim_result['func_vals']).cummin()
    display.clear_output(wait = True)
    min_ind = pd.Series(optim_result['func_vals']).argmin()
    print(paras, "=", optim_result['x_iters'][min_ind], pd.Series(optim_result['func_vals']))
    sns.lineplot(cv_values)
    plt.show()
```

```
gcv.fit(X_train_scaled, y_train, callback = monitor)
```

```
['n_neighbors', 'p', 'weights'] = [9, 1.0008321732366932, 'distance'] 5756.172382596493
```



```
BayesSearchCV(cv=KFold(n_splits=5, random_state=1, shuffle=True),
               estimator=KNeighborsRegressor(), n_iter=180, n_jobs=-1,
               random_state=10, scoring='neg_root_mean_squared_error',
               search_spaces={'n_neighbors': Integer(low=1, high=500, prior='uniform', transform='log'),
                              'p': Real(low=1, high=10, prior='uniform', transform='normalize'),
                              'weights': Categorical(categories=('uniform', 'distance'), prior='uniform')})
```

### 3.4 cross\_validate()

We have used `cross_val_score()` and `cross_val_predict()` so far.

When can we use one over the other?



The function `cross_validate()` is similar to `cross_val_score()` except that it has the option to return multiple cross-validated metrics, instead of a single one.

Consider the heart disease classification problem, where the response is **target** (*whether the person has a heart disease or not*).

```
data = pd.read_csv('Datasets/heart_disease_classification.csv')
data.head()
```

	age	sex	cp	trestbps	chol	fbs	restecg	thalach	exang	oldpeak	slope	ca	thal	target
0	63	1	3	145	233	1	0	150	0	2.3	0	0	1	1
1	37	1	2	130	250	0	1	187	0	3.5	0	0	2	1
2	41	0	1	130	204	0	0	172	0	1.4	2	0	2	1
3	56	1	1	120	236	0	1	178	0	0.8	2	0	2	1
4	57	0	0	120	354	0	1	163	1	0.6	2	0	2	1

Let us pre-process the data.

```
# First, separate the response and the predictors
y = data['target']
X = data.drop('target', axis=1)
```

```
# Separate the data (X,y) into training and test
```

```
# Inputs:
# data
# train-test ratio
# random_state for reproducible code
```

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=20, stratify=y)
```

```
# stratify=y makes sure the class 0 to class 1 ratio in the training and test sets are kept the same
```

```
model = KNeighborsClassifier()
sc = StandardScaler()
sc.fit(X_train)
X_train_scaled = sc.transform(X_train)
X_test_scaled = sc.transform(X_test)
```

Suppose we want to take recall above a certain threshold with the highest precision possible. `cross_validate()` computes the cross-validated score for multiple metrics - rest is the same as `cross_val_score()`.

```

Ks = np.arange(10,200,10)

scores = []

for K in Ks:
    model = KNeighborsClassifier(n_neighbors=K) # Keeping distance uniform
    scores.append(cross_validate(model, X_train_scaled, y_train, cv=5, scoring = ['accuracy'

scores

# The output is now a list of dicts - easy to convert to a df

df_scores = pd.DataFrame(scores) # We need to handle test_recall and test_precision cols

df_scores['CV_recall'] = df_scores['test_recall'].apply(np.mean)
df_scores['CV_precision'] = df_scores['test_precision'].apply(np.mean)
df_scores['CV_accuracy'] = df_scores['test_accuracy'].apply(np.mean)

df_scores.index = Ks # We can set K values as indices for convenience

#df_scores
# What happens as K increases?
# Recall increases (not monotonically)
# Precision decreases (not monotonically)
# Why?
# Check the class distribution in the data - more obs with class 1
# As K gets higher, the majority class overrules (visualized in the slides)
# More 1s means less FNs - higher recall
# More 1s means more FPs - lower precision
# Would this be the case for any dataset?
# NO!! Depends on what the majority class is!

```

Suppose we wish to have the maximum possible precision for at least 95% recall.

The optimal  $K$  will be:

```
df_scores.loc[df_scores['CV_recall'] > 0.95, 'CV_precision'].idxmax()
```

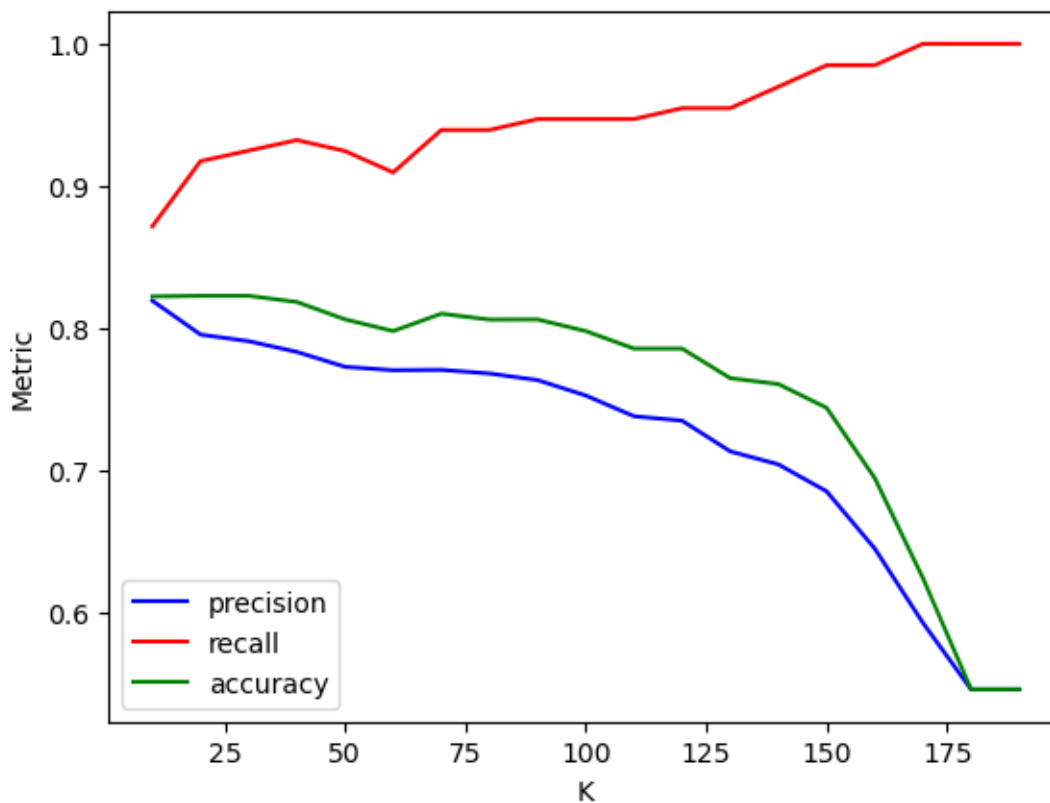
120

The cross-validated precision, recall and accuracy for the optimal  $K$  are:

```
df_scores.loc[120, ['CV_recall', 'CV_precision', 'CV_accuracy']]
```

```
CV_recall      0.954701  
CV_precision    0.734607  
CV_accuracy     0.785374  
Name: 120, dtype: object
```

```
sns.lineplot(x = df_scores.index, y = df_scores.CV_precision, color = 'blue', label = 'precision')  
sns.lineplot(x = df_scores.index, y = df_scores.CV_recall, color = 'red', label = 'recall')  
sns.lineplot(x = df_scores.index, y = df_scores.CV_accuracy, color = 'green', label = 'accuracy')  
plt.ylabel('Metric')  
plt.xlabel('K')  
plt.show()
```



## **Part II**

# **Tree based models**

## 4 Regression trees

*Read section 8.1.1 of the book before using these notes.*

*Note that in this course, lecture notes are not sufficient, you must read the book for better understanding. Lecture notes are just implementing the concepts of the book on a dataset, but not explaining the concepts elaborately.*

```
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.metrics import mean_squared_error, r2_score
from sklearn.model_selection import cross_val_score, train_test_split, KFold, RepeatedKFold,
GridSearchCV, ParameterGrid, RandomizedSearchCV
from sklearn.tree import DecisionTreeRegressor
from skopt import BayesSearchCV
from skopt.space import Integer, Categorical, Real
from IPython import display

#Libraries for visualizing trees
from sklearn.tree import export_graphviz, export_text
from six import StringIO
from IPython.display import Image
import pydotplus
import time as tm

#Using the same datasets as used for linear regression in STAT303-2,
#so that we can compare the non-linear models with linear regression
trainf = pd.read_csv('./Datasets/Car_features_train.csv')
trainp = pd.read_csv('./Datasets/Car_prices_train.csv')
testf = pd.read_csv('./Datasets/Car_features_test.csv')
testp = pd.read_csv('./Datasets/Car_prices_test.csv')
train = pd.merge(trainf, trainp)
test = pd.merge(testf, testp)
train.head()
```

	carID	brand	model	year	transmission	mileage	fuelType	tax	mpg	engineSize	price
0	18473	bmw	6 Series	2020	Semi-Auto	11	Diesel	145	53.3282	3.0	37980
1	15064	bmw	6 Series	2019	Semi-Auto	10813	Diesel	145	53.0430	3.0	33980
2	18268	bmw	6 Series	2020	Semi-Auto	6	Diesel	145	53.4379	3.0	36850
3	18480	bmw	6 Series	2017	Semi-Auto	18895	Diesel	145	51.5140	3.0	25998
4	18492	bmw	6 Series	2015	Automatic	62953	Diesel	160	51.4903	3.0	18990

## 4.1 Building a regression tree

Develop a regression tree to predict car price based on mileage

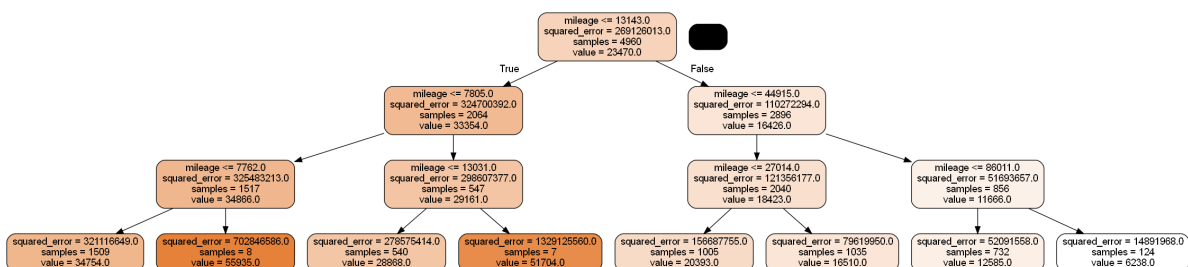
```
X = train['mileage']
y = train['price']
```

```
#Defining the object to build a regression tree
model = DecisionTreeRegressor(random_state=1, max_depth=3)

#Fitting the regression tree to the data
model.fit(X.values.reshape(-1,1), y)
```

```
DecisionTreeRegressor(max_depth=3, random_state=1)
```

```
#Visualizing the regression tree
dot_data = StringIO()
export_graphviz(model, out_file=dot_data,
                filled=True, rounded=True,
                feature_names = ['mileage'], precision=0)
graph = pydotplus.graph_from_dot_data(dot_data.getvalue())
graph.write_png('car_price_tree.png')
Image(graph.create_png())
```

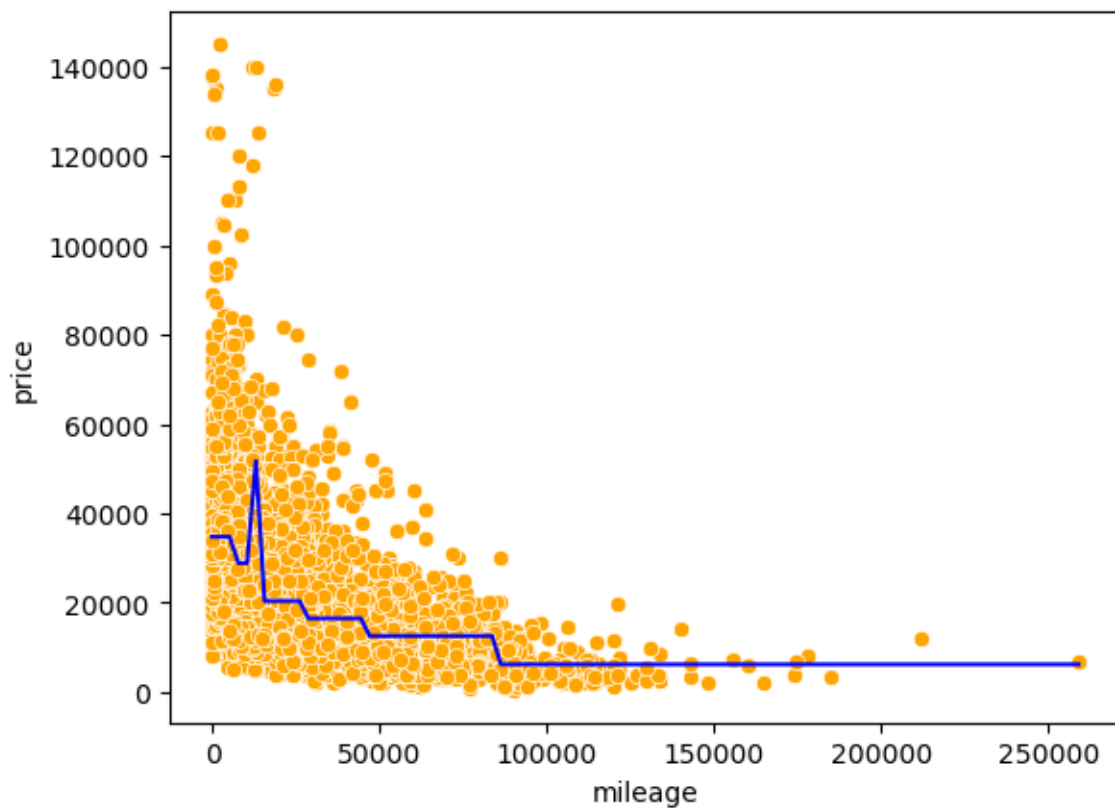


```
#prediction on test data
pred=model.predict(test[['mileage']].values)
```

```
#RMSE on test data
np.sqrt(mean_squared_error(test.price, pred))
```

13764.798425410803

```
#Visualizing the model fit
Xtest = np.linspace(min(X), max(X), 100)
pred_test = model.predict(Xtest.reshape(-1,1))
sns.scatterplot(x = 'mileage', y = 'price', data = train, color = 'orange')
sns.lineplot(x = Xtest, y = pred_test, color = 'blue');
```



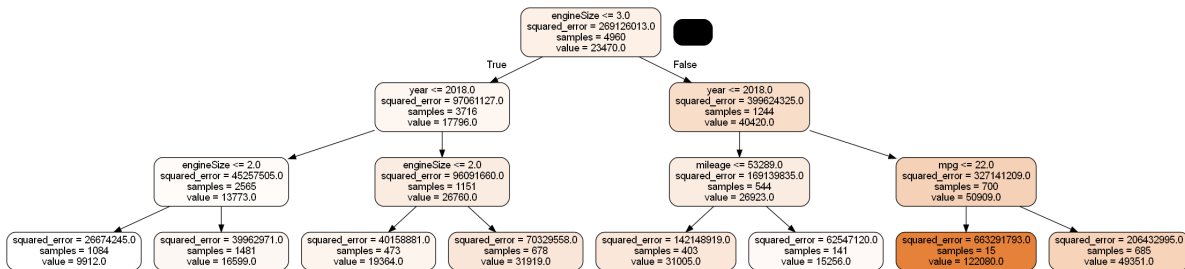
All cars falling within the same terminal node have the same predicted price, which is seen as flat line segments in the above model curve.

**Develop a regression tree to predict car price based on mileage, mpg, engineSize and year**

```

X = train[['mileage','mpg','year','engineSize']]
model = DecisionTreeRegressor(random_state=1, max_depth=3)
model.fit(X, y)
dot_data = StringIO()
export_graphviz(model, out_file=dot_data,
                filled=True, rounded=True,
                feature_names = ['mileage','mpg','year','engineSize'],precision=0)
graph = pydotplus.graph_from_dot_data(dot_data.getvalue())
graph.write_png('car_price_tree.png')
Image(graph.create_png())

```



The model can also be visualized in the text format as below.

```
print(export_text(model))
```

```

|--- feature_3 <= 2.75
|   |--- feature_2 <= 2018.50
|   |   |--- feature_3 <= 1.75
|   |   |   |--- value: [9912.24]
|   |   |   |--- feature_3 > 1.75
|   |   |   |   |--- value: [16599.03]
|   |   |--- feature_2 > 2018.50
|   |   |   |--- feature_3 <= 1.90
|   |   |   |   |--- value: [19363.81]
|   |   |   |   |--- feature_3 > 1.90
|   |   |   |   |   |--- value: [31919.42]
|--- feature_3 > 2.75
|   |--- feature_2 <= 2017.50
|   |   |--- feature_0 <= 53289.00
|   |   |   |--- value: [31004.63]
|   |   |   |--- feature_0 > 53289.00
|   |   |   |   |--- value: [15255.91]
|   |--- feature_2 > 2017.50

```



```
|   |   |--- feature_1 <= 21.79
|   |   |   |--- value: [122080.00]
|   |   |--- feature_1 > 21.79
|   |   |   |--- value: [49350.79]
```

## 4.2 Optimizing parameters to improve the regression tree

Let us find the optimal depth of the tree and the number of terminal nodes (leaves) by cross validation.

### 4.2.1 Range of hyperparameter values

First, we'll find the minimum and maximum possible values of the depth and leaves, and then find the optimal value in that range.

```
model = DecisionTreeRegressor(random_state=1)
model.fit(X, y)

print("Maximum tree depth =", model.get_depth())

print("Maximum leaves =", model.get_n_leaves())
```

```
Maximum tree depth = 29
Maximum leaves = 4845
```

### 4.2.2 Cross validation: Coarse grid

We'll use the `sklearn` function `GridSearchCV` to find the optimal hyperparameter values over a grid of possible values. By default, `GridSearchCV` returns the optimal hyperparameter values based on the coefficient of determination  $R^2$ . However, the `scoring` argument of the function can be used to find the optimal parameters based on several different criteria as mentioned in the [scoring-parameter documentation](#).

```
#Finding cross-validation error for trees
parameters = {'max_depth':range(2,30, 3),'max_leaf_nodes':range(2,4900, 100)}
cv = KFold(n_splits = 5,shuffle=True,random_state=1)
model = GridSearchCV(DecisionTreeRegressor(random_state=1), parameters, n_jobs=-1,verbose=1,
model.fit(X, y)
print (model.best_score_, model.best_params_)
```

Fitting 5 folds for each of 490 candidates, totalling 2450 fits  
0.8433100904754441 {'max\_depth': 11, 'max\_leaf\_nodes': 302}

Let us find the optimal hyperparameters based on root mean squared error (RMSE), instead of  $R^2$ . Let us compute  $R^2$  as well during cross validation, as we can compute multiple performance metrics using the `scoring` argument. However, when computing multiple performance metrics, we will need to specify the performance metric used to find the optimal hyperparameters with the `refit` argument.

```
#Finding cross-validation error for trees
parameters = {'max_depth':range(2,30, 3),'max_leaf_nodes':range(2,4900, 100)}
cv = KFold(n_splits = 5,shuffle=True,random_state=1)
model = GridSearchCV(DecisionTreeRegressor(random_state=1), parameters, n_jobs=-1,verbose=1,
                     scoring=['neg_root_mean_squared_error', 'r2'], refit = 'neg_root_mean_squared_error')
model.fit(X, y)
print (model.best_score_, model.best_params_)
```

Fitting 5 folds for each of 490 candidates, totalling 2450 fits  
-6475.329183576911 {'max\_depth': 11, 'max\_leaf\_nodes': 302}

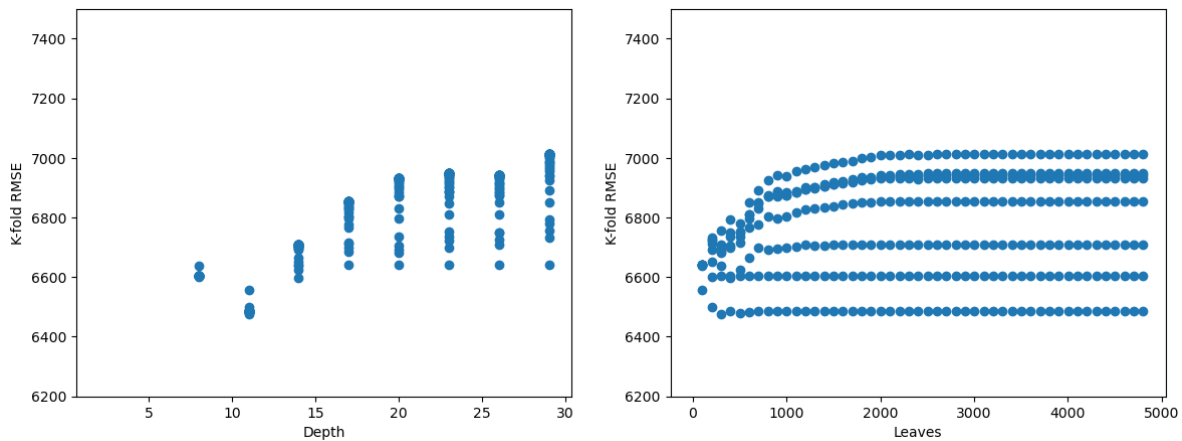
Note that as the `GridSearchCV` function maximizes the performance metric to find the optimal hyperparameters, we are maximizing the negative root mean squared error (`neg_root_mean_squared_error`), and the function returns the optimal negative mean squared error.

Let us visualize the mean squared error based on the hyperparameter values. We'll use the cross validation results stored in the `cv_results_` attribute of the `GridSearchCV` `fit()` object.

```
#Detailed results of k-fold cross validation
cv_results = pd.DataFrame(model.cv_results_)
cv_results.head()
```

	mean_fit_time	std_fit_time	mean_score_time	std_score_time	param_max_depth	param_max
0	0.010178	7.531409e-04	0.003791	0.000415	2	2
1	0.009574	1.758238e-03	0.003782	0.000396	2	102
2	0.009774	7.458305e-04	0.003590	0.000488	2	202
3	0.009568	4.953541e-04	0.003391	0.000489	2	302
4	0.008976	6.843901e-07	0.003192	0.000399	2	402

```
fig, axes = plt.subplots(1,2,figsize=(14,5))
plt.subplots_adjust(wspace=0.2)
axes[0].plot(cv_results.param_max_depth, (-cv_results.mean_test_neg_root_mean_squared_error))
axes[0].set_ylim([6200, 7500])
axes[0].set_xlabel('Depth')
axes[0].set_ylabel('K-fold RMSE')
axes[1].plot(cv_results.param_max_leaf_nodes, (-cv_results.mean_test_neg_root_mean_squared_error))
axes[1].set_ylim([6200, 7500])
axes[1].set_xlabel('Leaves')
axes[1].set_ylabel('K-fold RMSE');
```



We observe that for a depth of around 8-14, and number of leaves within 1000, we get the lowest  $K$ -fold RMSE. So, we should do a finer search in that region to obtain more precise hyperparameter values.

### 4.2.3 Cross validation: Finer grid

```
#Finding cross-validation error for trees
start_time = tm.time()
parameters = {'max_depth':range(8,15),'max_leaf_nodes':range(2,1000)}
cv = KFold(n_splits = 5,shuffle=True,random_state=1)
model = GridSearchCV(DecisionTreeRegressor(random_state=1), parameters, n_jobs=-1,verbose=1,
                    scoring = 'neg_root_mean_squared_error')
model.fit(X, y)
print (model.best_score_, model.best_params_)
print("Time taken =", round((tm.time() - start_time)/60), "minutes")
```

```
Fitting 5 folds for each of 6986 candidates, totalling 34930 fits
-6414.468922119372 {'max_depth': 10, 'max_leaf_nodes': 262}
Time taken = 2 minutes
```

From the above cross-validation, the optimal hyperparameter values are `max_depth = 10` and `max_leaf_nodes = 262`. Note that the cross-validation score with finer grid is only slightly lower than the course grid. However, depending on the dataset, the finer grid may lead to more benefit.

```
#Developing the tree based on optimal hyperparameters found by cross-validation
model = DecisionTreeRegressor(random_state=1, max_depth=10,max_leaf_nodes=262)
model.fit(X, y)
```

```
DecisionTreeRegressor(max_depth=10, max_leaf_nodes=262, random_state=1)
```

```
#RMSE on test data
Xtest = test[['mileage','mpg','year','engineSize']]
np.sqrt(mean_squared_error(test.price, model.predict(Xtest)))
```

```
6921.0404660552895
```

The RMSE for the decision tree is lower than that of linear regression models with these four predictors. This may be probably due to car price having a highly non-linear association with the predictors.

Note that we may also use `RandomizedSearchCV()` or `BayesSearchCV()` to optimize the hyperparameters.

**Predictor importance:** The importance of a predictor is computed as the (normalized) total reduction of the criterion (SSE in case of regression trees) brought by that predictor.

Warning: impurity-based feature importances can be misleading for high cardinality features (many unique values) *Source: <https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeRegressor.html>*

### Why?

Because high cardinality predictors will tend to overfit. When the predictors have high cardinality, it means they form little groups (*in the leaf nodes*) and then the model “learns” the individuals, instead of “learning” the general trend. The higher the cardinality of the predictor, the more prone is the model to overfitting.

```
model.feature_importances_
```

```
array([0.04490344, 0.15882336, 0.29739951, 0.49887369])
```

*Engine size* is the most important predictor, followed by *year*, which is followed by *mpg*, and *mileage* is the least important predictor.

## 4.3 Cost complexity pruning

While optimizing parameters above, we optimized them within a range that we thought was reasonable. While doing so, we restricted ourselves to considering only a subset of the unpruned tree. Thus, we could have missed out on finding the optimal tree (or the best model).

With cost complexity pruning, we first develop an unpruned tree without any restrictions. Then, using cross validation, we find the optimal value of the tuning parameter  $\alpha$ . All the non-terminal nodes for which  $\alpha_{eff}$  is smaller than the optimal  $\alpha$  will be pruned. You will need to check out the link below to understand this better.

Check out a detailed explanation of how cost complexity pruning is implemented in sklearn at: <https://scikit-learn.org/stable/modules/tree.html#minimal-cost-complexity-pruning>

Here are some informative visualizations that will help you understand what is happening in cost complexity pruning: [https://scikit-learn.org/stable/auto\\_examples/tree/plot\\_cost\\_complexity\\_pruning.html#sphx-glr-auto-examples-tree-plot-cost-complexity-pruning-py](https://scikit-learn.org/stable/auto_examples/tree/plot_cost_complexity_pruning.html#sphx-glr-auto-examples-tree-plot-cost-complexity-pruning-py)

```
model = DecisionTreeRegressor(random_state = 1)#model without any restrictions
path= model.cost_complexity_pruning_path(X,y)# Compute the pruning path during Minimal Cost-
```

```
alphas=path['ccp_alphas']
```

```
len(alphas)
```

```
4126
```

```
start_time = tm.time()
cv = KFold(n_splits = 5,shuffle=True,random_state=1)
tree = GridSearchCV(DecisionTreeRegressor(random_state=1), param_grid = {'ccp_alpha':alphas},
                    scoring = 'neg_mean_squared_error',n_jobs=-1,verbose=1,cv=cv)
tree.fit(X, y)
print (tree.best_score_, tree.best_params_)
print("Time taken =",round((tm.time()-start_time)/60), "minutes")
```

```
Fitting 5 folds for each of 4126 candidates, totalling 20630 fits
-44150619.209031895 {'ccp_alpha': 143722.94076639024}
Time taken = 2 minutes
```

The code took 2 minutes to run on a dataset of about 5000 observations and 4 predictors.

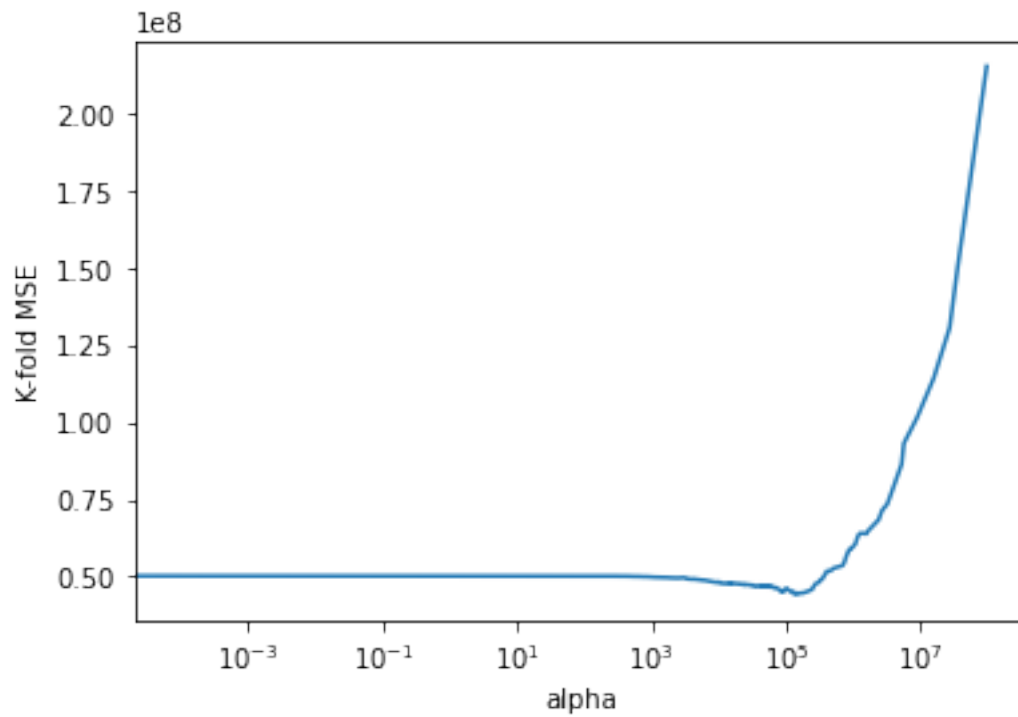
```
model = DecisionTreeRegressor(ccp_alpha=143722.94076639024,random_state=1)
model.fit(X, y)
pred = model.predict(Xtest)
np.sqrt(mean_squared_error(test.price, pred))
```

```
7306.592294294368
```

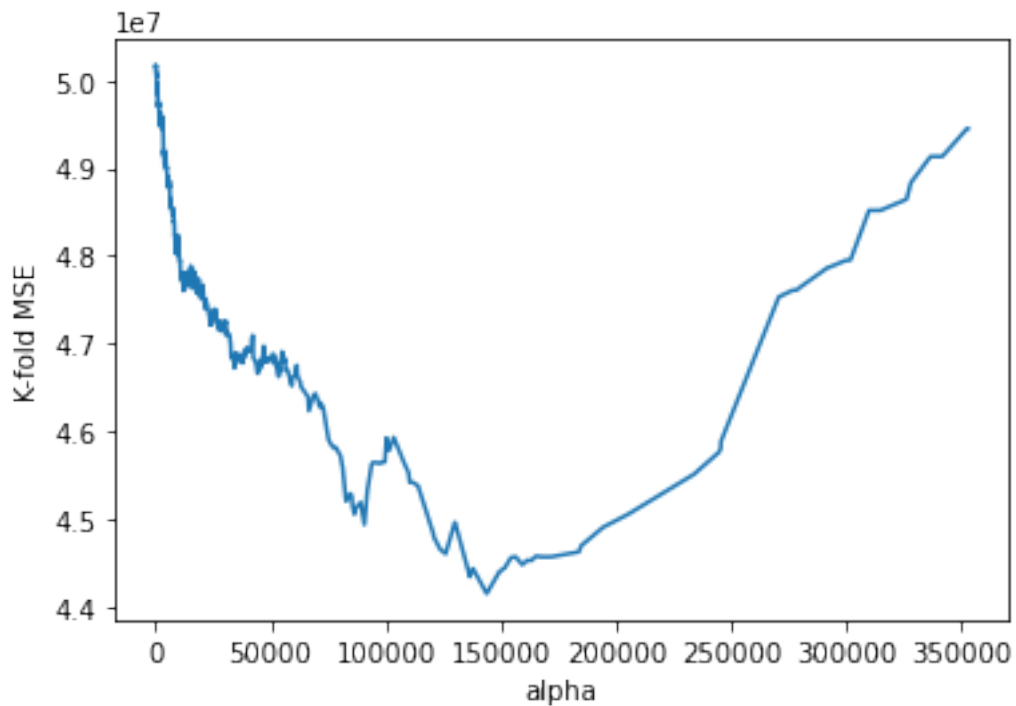
The RMSE for the decision tree with cost complexity pruning is lower than that of linear regression models and spline regression models (including MARS), with these four predictors. However, it is higher than the one obtained with tuning tree parameters using grid search (shown previously). Cost complexity pruning considers a completely unpruned tree unlike the ‘grid search’ method of searching over a grid of hyperparameters such as `max_depth` and `max_leaf_nodes`, and thus may seem to be more comprehensive than the ‘grid search’ approach. However, both the approaches may consider trees that are not considered by the other approach, and thus either one may provide a more accurate model. Depending on the grid of parameters chosen for cross validation, the grid search method may be more or less comprehensive than cost complexity pruning.

```
gridcv_results = pd.DataFrame(tree.cv_results_)
cv_error = -gridcv_results['mean_test_score']
```

```
#Visualizing the 5-fold cross validation error vs alpha
plt.plot(alphas,cv_error)
plt.xscale('log')
plt.xlabel('alpha')
plt.ylabel('K-fold MSE');
```



```
#Zooming in the above visualization to see the alpha where the 5-fold cross validation error  
plt.plot(alphas[0:4093],cv_error[0:4093])  
plt.xlabel('alpha')  
plt.ylabel('K-fold MSE');
```



#### 4.3.1 Depth vs alpha; Node counts vs alpha

```

time = time.time()
trees=[]
for i in alphas:
    tree = DecisionTreeRegressor(ccp_alpha=i,random_state=1)
    tree.fit(X, train['price'])
    trees.append(tree)
print(time.time()-stime)

```

268.10325384140015

This code takes 4.5 minutes to run

```

node_counts = [clf.tree_.node_count for clf in trees]
depth = [clf.tree_.max_depth for clf in trees]

```

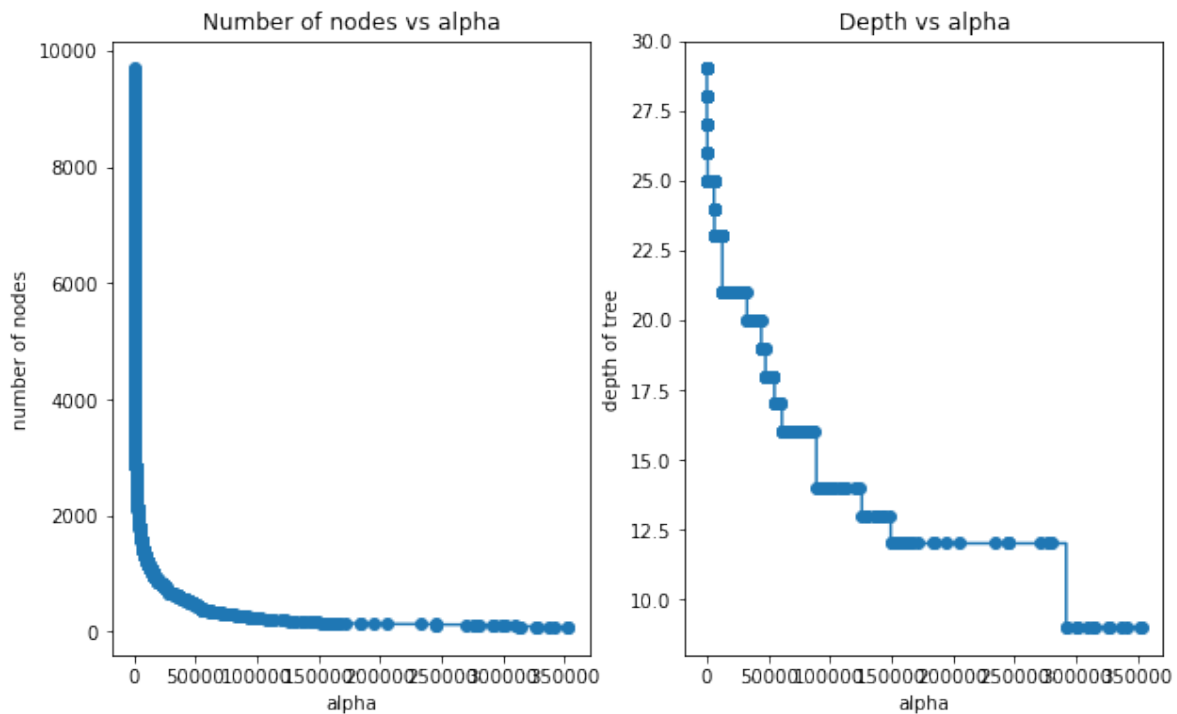


```

fig, ax = plt.subplots(1, 2, figsize=(10,6))
ax[0].plot(alphas[0:4093], node_counts[0:4093], marker="o", drawstyle="steps-post")#Plotting
ax[0].set_xlabel("alpha")
ax[0].set_ylabel("number of nodes")
ax[0].set_title("Number of nodes vs alpha")
ax[1].plot(alphas[0:4093], depth[0:4093], marker="o", drawstyle="steps-post")#Plotting the z
ax[1].set_xlabel("alpha")
ax[1].set_ylabel("depth of tree")
ax[1].set_title("Depth vs alpha")
fig.tight_layout()

```

Text(0.5, 1.0, 'Depth vs alpha')



#### 4.3.2 Train and test accuracies (R-squared) vs alpha

```

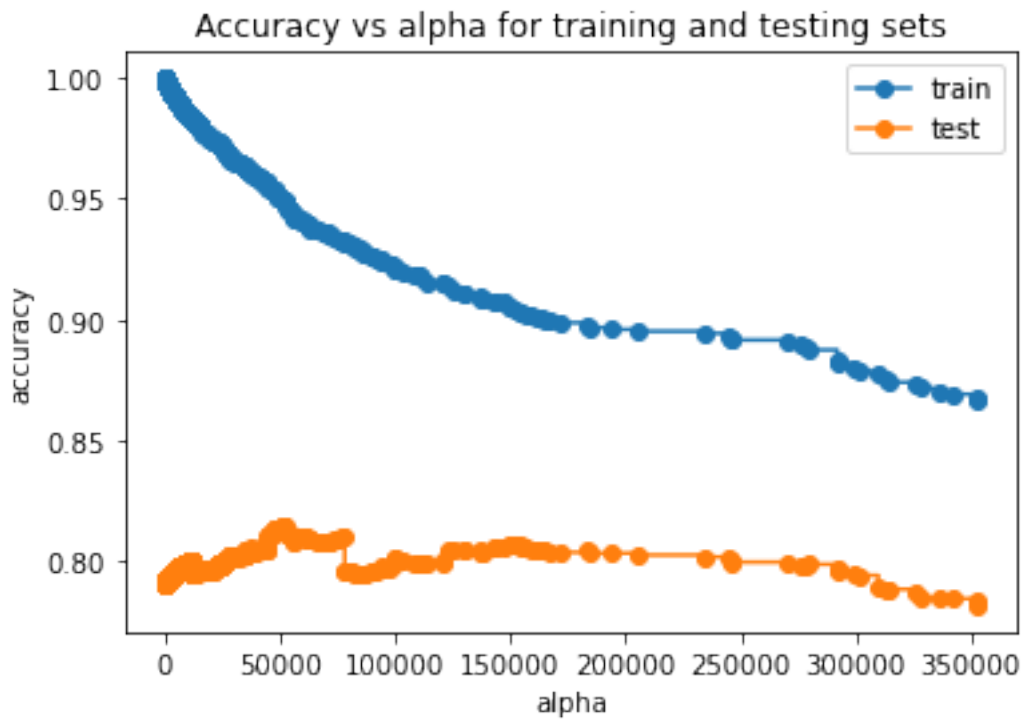
train_scores = [clf.score(X, y) for clf in trees]
test_scores = [clf.score(Xtest, test.price) for clf in trees]

```

```

fig, ax = plt.subplots()
ax.set_xlabel("alpha")
ax.set_ylabel("accuracy")
ax.set_title("Accuracy vs alpha for training and testing sets")
ax.plot(alphas[0:4093], train_scores[0:4093], marker="o", label="train", drawstyle="steps-post")
ax.plot(alphas[0:4093], test_scores[0:4093], marker="o", label="test", drawstyle="steps-post")
ax.legend()
plt.show()

```



## 5 Classification trees

*Read section 8.1.2 of the book before using these notes.*

*Note that in this course, lecture notes are not sufficient, you must read the book for better understanding. Lecture notes are just implementing the concepts of the book on a dataset, but not explaining the concepts elaborately.*

```
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import cross_val_score, train_test_split, cross_val_predict
from sklearn.metrics import roc_curve, precision_recall_curve, auc, make_scorer, recall_score
from sklearn.model_selection import StratifiedKFold, KFold
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import GridSearchCV

#Libraries for visualizing trees
from sklearn.tree import export_graphviz
from six import StringIO
from IPython.display import Image
import pydotplus

import time as time
```

```
train = pd.read_csv('./Datasets/diabetes_train.csv')
test = pd.read_csv('./Datasets/diabetes_test.csv')
```

```
test.head()
```

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age
0	6	148	72	35	0	33.6	0.627	50
1	2	197	70	45	543	30.5	0.158	53

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age
2	1	115	70	30	96	34.6	0.529	32
3	8	99	84	0	0	35.4	0.388	50
4	7	147	76	0	0	39.4	0.257	43

## 5.1 Building a classification tree

Develop a classification tree to predict if a person has diabetes.

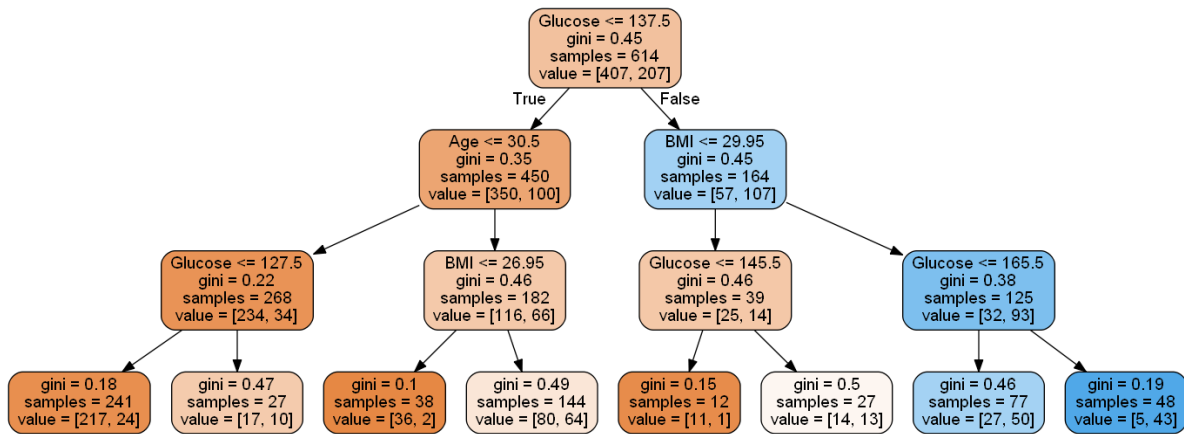
```
X = train.drop(columns = 'Outcome')
Xtest = test.drop(columns = 'Outcome')
y = train['Outcome']
ytest = test['Outcome']
```

```
#Defining the object to build a classification tree
model = DecisionTreeClassifier(random_state=1, max_depth=3)

#Fitting the regression tree to the data
model.fit(X, y)
```

```
DecisionTreeClassifier(max_depth=3, random_state=1)
```

```
#Visualizing the regression tree
dot_data = StringIO()
export_graphviz(model, out_file=dot_data,
                filled=True, rounded=True,
                feature_names =X.columns,precision=2)
graph = pydotplus.graph_from_dot_data(dot_data.getvalue())
#graph.write_png('car_price_tree.png')
Image(graph.create_png())
```



```
# Performance metrics computation
```

```
#Computing the accuracy
```

```
y_pred = model.predict(Xtest)
print("Accuracy: ",accuracy_score(y_pred, ytest)*100)
```

```
#Computing the ROC-AUC
```

```
y_pred_prob = model.predict_proba(Xtest)[: ,1]
fpr, tpr, auc_thresholds = roc_curve(ytest, y_pred_prob)
print("ROC-AUC: ",auc(fpr, tpr))# AUC of ROC
```

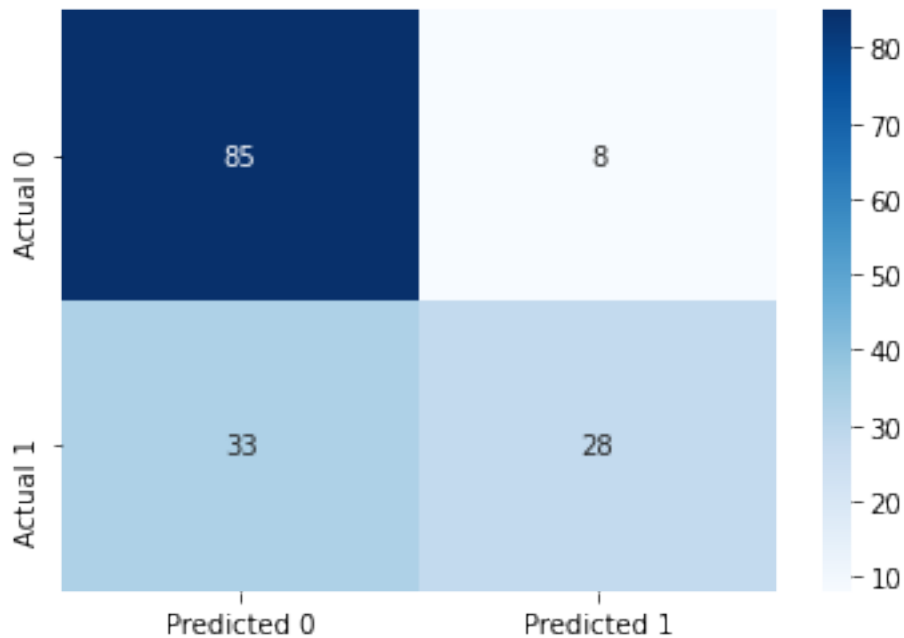
```
#Computing the precision and recall
```

```
print("Precision: ", precision_score(ytest, y_pred))
print("Recall: ", recall_score(ytest, y_pred))
```

```
#Confusion matrix
```

```
cm = pd.DataFrame(confusion_matrix(ytest, y_pred), columns=['Predicted 0', 'Predicted 1'],
                  index = ['Actual 0', 'Actual 1'])
sns.heatmap(cm, annot=True, cmap='Blues', fmt='g');
```

```
Accuracy: 73.37662337662337
ROC-AUC: 0.8349197955226512
Precision: 0.7777777777777778
Recall: 0.45901639344262296
```



## 5.2 Optimizing hyperparameters to optimize performance

In case of diabetes, it is important to reduce FNR (False negative rate) or maximize recall. This is because if a person has diabetes, the consequences of predicting that they don't have diabetes can be much worse than the other way round.

Let us find the optimal depth of the tree and the number of terminal nodes (leaves) that minimizes the FNR or maximizes recall.

Find the maximum values of depth and number of leaves.

```
#Defining the object to build a regression tree
model = DecisionTreeClassifier(random_state=1)

#Fitting the regression tree to the data
model.fit(X, y)
```

```
DecisionTreeClassifier(random_state=1)
```

```
# Maximum number of leaves
model.get_n_leaves()
```

118

```
# Maximum depth
model.get_depth()
```

14

```
#Defining parameters and the range of values over which to optimize
param_grid = {
    'max_depth': range(2,14),
    'max_leaf_nodes': range(2,118),
    'max_features': range(1, 9)
}
```

```
#Grid search to optimize parameter values
```

```
start_time = time.time()
skf = StratifiedKFold(n_splits=5)#The folds are made by preserving the percentage of samples

#Minimizing FNR is equivalent to maximizing recall
grid_search = GridSearchCV(DecisionTreeClassifier(random_state=1), param_grid, scoring=['pre
                                refit="recall", cv=skf, n_jobs=-1, verbose = True)
grid_search.fit(X, y)

# make the predictions
y_pred = grid_search.predict(Xtest)

print('Train accuracy : %.3f'%grid_search.best_estimator_.score(X, y))
print('Test accuracy : %.3f'%grid_search.best_estimator_.score(Xtest, ytest))
print('Best recall Through Grid Search : %.3f'%grid_search.best_score_)

print('Best params for recall')
print(grid_search.best_params_)

print("Time taken =", round((time.time() - start_time)), "seconds")
```

Fitting 5 folds for each of 11136 candidates, totalling 55680 fits

Train accuracy : 0.785

Test accuracy : 0.675

Best recall Through Grid Search : 0.658

Best params for recall

{'max\_depth': 4, 'max\_features': 2, 'max\_leaf\_nodes': 8}

Time taken = 70 seconds

## 5.3 Optimizing the decision threshold probability

Note that decision threshold probability is not tuned with `GridSearchCV` because `GridSearchCV` is a technique used for hyperparameter tuning in machine learning models, and the decision threshold probability is not a hyperparameter of the model.

The decision threshold is set to 0.5 by default during hyperparameter tuning with `GridSearchCV`.

`GridSearchCV` is used to tune hyperparameters that control the internal settings of a machine learning model, such as learning rate, regularization strength, and maximum tree depth, among others. These hyperparameters affect the model's internal behavior and performance. On the other hand, the decision threshold is an external parameter that is used to interpret the model's output and make predictions based on the predicted probabilities.

To tune the decision threshold, one typically needs to manually adjust it after the model has been trained and evaluated using a specific set of hyperparameter values. This can be done using methods, which involve evaluating the model's performance at different decision threshold values and selecting the one that best meets the desired trade-off between false positives and false negatives based on the specific problem requirements.

As the recall will always be 100% for a decision threshold probability of zero, we'll find a decision threshold probability that balances recall with another performance metric such as precision, false positive rate, accuracy, etc. Below are a couple of examples that show we can balance recall with (1) precision or (2) false positive rate.

### 5.3.1 Balancing recall with precision

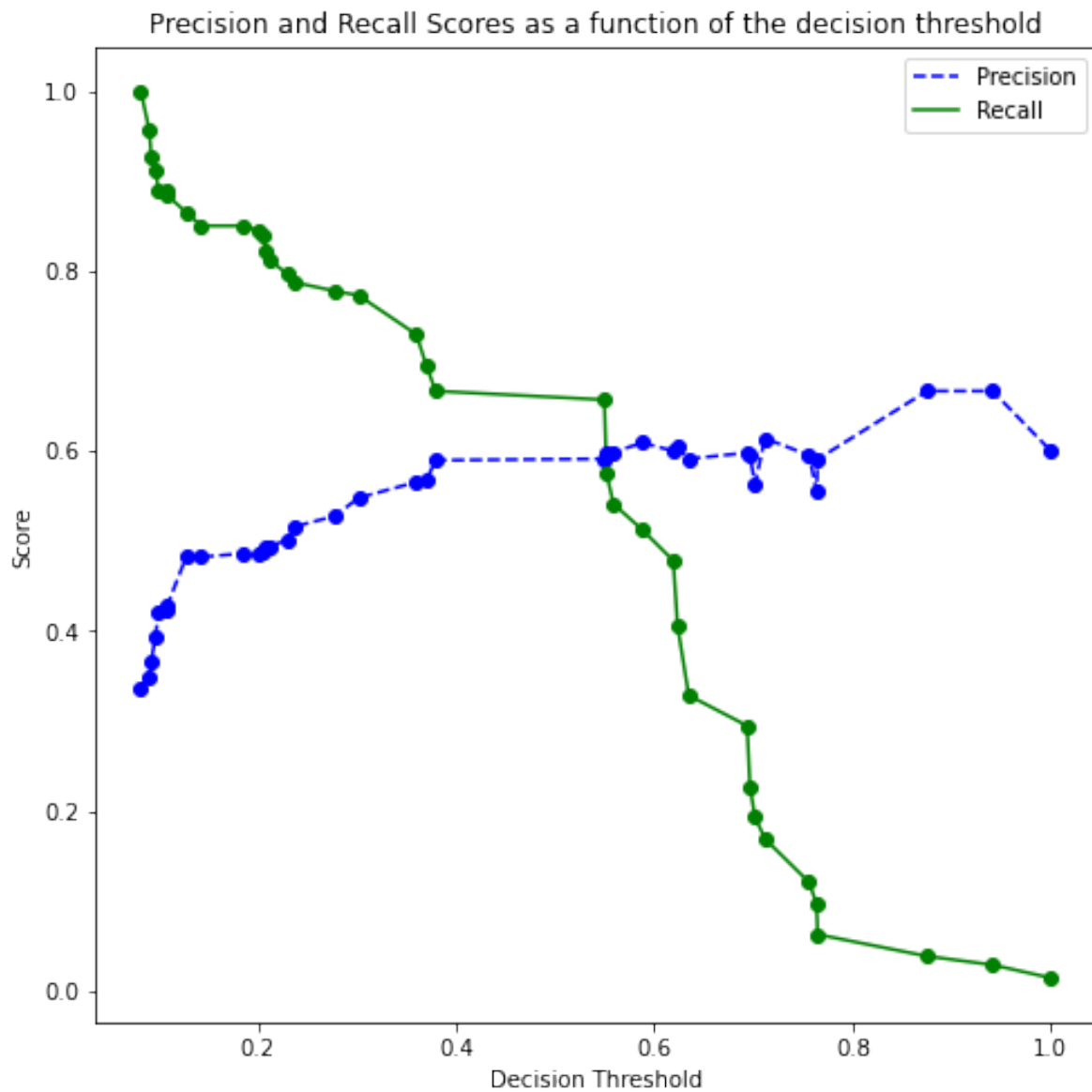
We can find a threshold probability that balances recall with precision.

```
model = DecisionTreeClassifier(random_state=1, max_depth = 4, max_leaf_nodes=8, max_features=
# Note that we are using the cross-validated predicted probabilities, instead of directly us
# predicted probabilities on train data, as the model may be overfitting on the train data, a
# may lead to misleading results
cross_val_ypred = cross_val_predict(DecisionTreeClassifier(random_state=1, max_depth = 4,
                                                            max_leaf_nodes=8, max_features=2), X
                                   y, cv = 5, method = 'predict_proba')

p, r, thresholds = precision_recall_curve(y, cross_val_ypred[:,1])
def plot_precision_recall_vs_threshold(precisions, recalls, thresholds):
    plt.figure(figsize=(8, 8))
    plt.title("Precision and Recall Scores as a function of the decision threshold")
```



```
plt.plot(thresholds, precisions[:-1], "b--", label="Precision")
plt.plot(thresholds, recalls[:-1], "g-", label="Recall")
plt.plot(thresholds, precisions[:-1], "o", color = 'blue')
plt.plot(thresholds, recalls[:-1], "o", color = 'green')
plt.ylabel("Score")
plt.xlabel("Decision Threshold")
plt.legend(loc='best')
plt.legend()
plot_precision_recall_vs_threshold(p, r, thresholds)
```



```
# Thresholds with precision and recall
np.concatenate([thresholds.reshape(-1,1), p[:-1].reshape(-1,1), r[:-1].reshape(-1,1)], axis =
```

```
array([[0.08196721, 0.33713355, 1.          ],
       [0.09045226, 0.34982332, 0.95652174],
       [0.09248555, 0.36641221, 0.92753623],
       [0.0964467 , 0.39293139, 0.91304348],
       [0.1         , 0.42105263, 0.88888889],
```

```
[0.10810811, 0.42298851, 0.88888889],
[0.10869565, 0.42857143, 0.88405797],
[0.12820513, 0.48378378, 0.8647343 ],
[0.14285714, 0.48219178, 0.85024155],
[0.18518519, 0.48618785, 0.85024155],
[0.2          , 0.48611111, 0.84541063],
[0.20512821, 0.48876404, 0.84057971],
[0.20833333, 0.49418605, 0.82125604],
[0.21276596, 0.49411765, 0.8115942 ],
[0.22916667, 0.50151976, 0.79710145],
[0.23684211, 0.51582278, 0.78743961],
[0.27777778, 0.52786885, 0.77777778],
[0.3015873 , 0.54794521, 0.77294686],
[0.36          , 0.56554307, 0.7294686 ],
[0.3697479 , 0.56692913, 0.69565217],
[0.37931034, 0.58974359, 0.66666667],
[0.54954955, 0.59130435, 0.65700483],
[0.55172414, 0.59798995, 0.57487923],
[0.55882353, 0.59893048, 0.5410628 ],
[0.58823529, 0.6091954 , 0.51207729],
[0.61904762, 0.6          , 0.47826087],
[0.62337662, 0.60431655, 0.4057971 ],
[0.63461538, 0.59130435, 0.32850242],
[0.69354839, 0.59803922, 0.29468599],
[0.69642857, 0.59493671, 0.22705314],
[0.70149254, 0.56338028, 0.19323671],
[0.71153846, 0.61403509, 0.16908213],
[0.75609756, 0.5952381 , 0.12077295],
[0.76363636, 0.55555556, 0.09661836],
[0.76470588, 0.59090909, 0.06280193],
[0.875          , 0.66666667, 0.03864734],
[0.94117647, 0.66666667, 0.02898551],
[1.          , 0.6          , 0.01449275]])
```

Suppose, we wish to have at least 80% recall, with the highest possible precision. Then, based on the precision-recall curve (*or the table above*), we should have a decision threshold probability of 0.21.

Let's assess the model's performance on test data with a threshold probability of 0.21.

```
# Performance metrics computation for the optimum decision threshold probability
desired_threshold = 0.21
```

```

y_pred_prob = model.predict_proba(Xtest)[:,-1]

# Classifying observations in the positive class (y = 1) if the predicted probability is greater
# than the desired decision threshold probability
y_pred = y_pred_prob > desired_threshold
y_pred = y_pred.astype(int)

#Computing the accuracy
print("Accuracy: ",accuracy_score(y_pred, ytest)*100)

#Computing the ROC-AUC
fpr, tpr, auc_thresholds = roc_curve(ytest, y_pred_prob)
print("ROC-AUC: ",auc(fpr, tpr))# AUC of ROC

#Computing the precision and recall
print("Precision: ", precision_score(ytest, y_pred))
print("Recall: ", recall_score(ytest, y_pred))

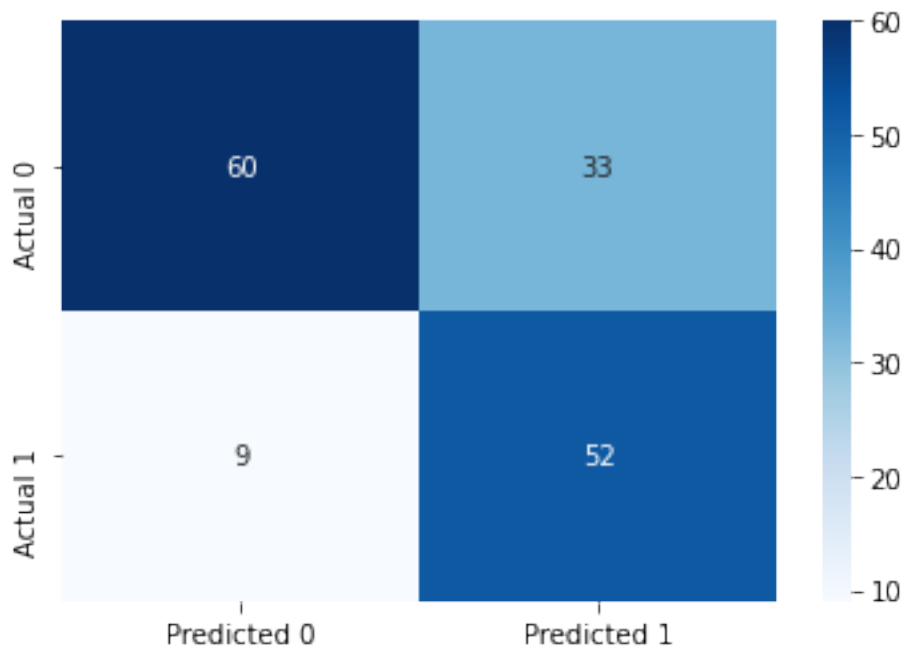
#Confusion matrix
cm = pd.DataFrame(confusion_matrix(ytest, y_pred),
                  columns=['Predicted 0', 'Predicted 1'], index = ['Actual 0', 'Actual 1'])
sns.heatmap(cm, annot=True, cmap='Blues', fmt='g');

```

```

Accuracy: 72.72727272727273
ROC-AUC: 0.7544509078089194
Precision: 0.611764705882353
Recall: 0.8524590163934426

```



### 5.3.2 Balancing recall with false positive rate

Suppose we wish to balance recall with false positive rate. We can optimize the model to maximize ROC-AUC, and then choose a point on the ROC-curve that balances recall with the false positive rate.

```
# Defining parameters and the range of values over which to optimize
param_grid = {
    'max_depth': range(2,14),
    'max_leaf_nodes': range(2,118),
    'max_features': range(1, 9)
}
```

```
#Grid search to optimize parameter values
```

```
start_time = time.time()
```

```
skf = StratifiedKFold(n_splits=5)#The folds are made by preserving the percentage of samples
```

```
#Minimizing FNR is equivalent to maximizing recall
```

```
grid_search = GridSearchCV(DecisionTreeClassifier(random_state=1), param_grid, scoring=['pre
    'roc_auc'], refit="roc_auc", cv=skf, n_jobs=-1, verbose = True)
```

```
grid_search.fit(X, y)
```

```
# make the predictions
y_pred = grid_search.predict(Xtest)

print('Best params for recall')
print(grid_search.best_params_)

print("Time taken =", round((time.time() - start_time)), "seconds")
```

Fitting 5 folds for each of 11136 candidates, totalling 55680 fits  
 Best params for recall  
 {'max\_depth': 6, 'max\_features': 2, 'max\_leaf\_nodes': 9}  
 Time taken = 72 seconds

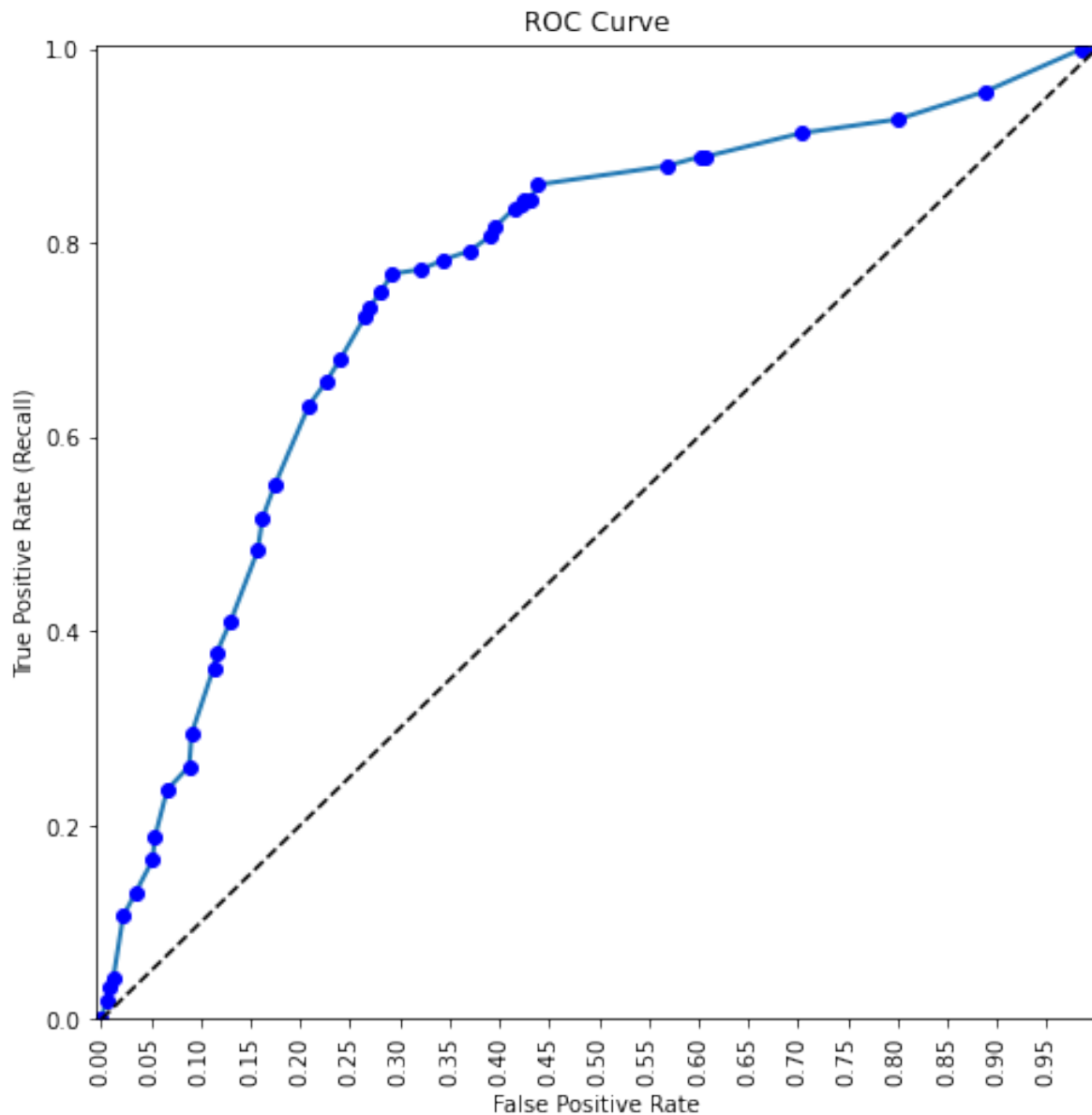
```
model = DecisionTreeClassifier(random_state=1, max_depth = 6, max_leaf_nodes=9, max_features=
```

```
cross_val_ypred = cross_val_predict(DecisionTreeClassifier(random_state=1, max_depth = 6,
                                                           max_leaf_nodes=9, max_features=2)
                                   y, cv = 5, method = 'predict_proba')
```

```
fpr, tpr, auc_thresholds = roc_curve(y, cross_val_ypred[:,1])
print(auc(fpr, tpr))# AUC of ROC
def plot_roc_curve(fpr, tpr, label=None):
    plt.figure(figsize=(8,8))
    plt.title('ROC Curve')
    plt.plot(fpr, tpr, linewidth=2, label=label)
    plt.plot(fpr, tpr, 'o', color = 'blue')
    plt.plot([0, 1], [0, 1], 'k--')
    plt.axis([-0.005, 1, 0, 1.005])
    plt.xticks(np.arange(0,1, 0.05), rotation=90)
    plt.xlabel("False Positive Rate")
    plt.ylabel("True Positive Rate (Recall)")

fpr, tpr, auc_thresholds = roc_curve(y, cross_val_ypred[:,1])
plot_roc_curve(fpr, tpr)
```

0.7605075431162388



```
# Thresholds with TPR and FPR
all_thresholds = np.concatenate([auc_thresholds.reshape(-1,1), tpr.reshape(-1,1), fpr.reshape(-1,1)])
recall_more_than_80 = all_thresholds[all_thresholds[:,1]>0.8,:]
# As the values in 'recall_more_than_80' are arranged in increasing order of recall and decreasing order of FPR,
# the first value will provide the maximum threshold probability for the recall to be more than 80%
# We wish to find the maximum threshold probability to obtain the minimum possible FPR
recall_more_than_80[0]
```

```
array([0.21276596, 0.80676329, 0.39066339])
```

Suppose, we wish to have at least 80% recall, with the lowest possible precision. Then, based on the ROC-AUC curve, we should have a decision threshold probability of 0.21.

Let's assess the model's performance on test data with a threshold probability of 0.21.

```
# Performance metrics computation for the optimum decision threshold probability
desired_threshold = 0.21

y_pred_prob = model.predict_proba(Xtest)[:,-1]

# Classifying observations in the positive class (y = 1) if the predicted probability is greater
# than the desired decision threshold probability
y_pred = y_pred_prob > desired_threshold
y_pred = y_pred.astype(int)

#Computing the accuracy
print("Accuracy: ",accuracy_score(y_pred, ytest)*100)

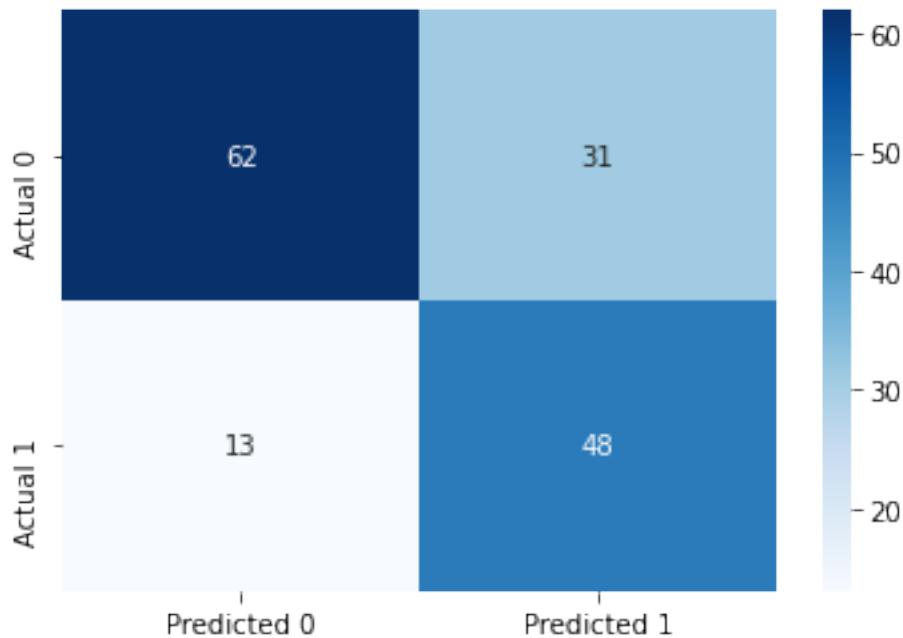
#Computing the ROC-AUC
fpr, tpr, auc_thresholds = roc_curve(ytest, y_pred_prob)
print("ROC-AUC: ",auc(fpr, tpr))# AUC of ROC

#Computing the precision and recall
print("Precision: ", precision_score(ytest, y_pred))
print("Recall: ", recall_score(ytest, y_pred))

#Confusion matrix
cm = pd.DataFrame(confusion_matrix(ytest, y_pred),
                  columns=['Predicted 0', 'Predicted 1'], index = ['Actual 0', 'Actual 1'])
sns.heatmap(cm, annot=True, cmap='Blues', fmt='g');
```

```
Accuracy: 71.42857142857143
ROC-AUC: 0.7618543980257358
Precision: 0.6075949367088608
Recall: 0.7868852459016393
```





## 5.4 Cost complexity pruning

Just as we did cost complexity pruning in a regression tree, we can do it to optimize the model for a classification tree.

```
model = DecisionTreeClassifier(random_state = 1)#model without any restrictions
path= model.cost_complexity_pruning_path(X,y)# Compute the pruning path during Minimal Cost-
```

```
alphas=path['ccp_alphas']
len(alphas)
```

58

```
#Grid search to optimize parameter values

skf = StratifiedKFold(n_splits=5)
grid_search = GridSearchCV(DecisionTreeClassifier(random_state = 1), param_grid = {'ccp_alpha':
scoring=['precision','recall','accuracy'],
refit="recall", cv=skf, n_jobs=-1, verbose=10)

grid_search.fit(X, y)
```

```
# make the predictions
y_pred = grid_search.predict(Xtest)

print('Best params for recall')
print(grid_search.best_params_)
```

Fitting 5 folds for each of 58 candidates, totalling 290 fits  
Best params for recall  
{'ccp\_alpha': 0.010561291712538737}

```
# Model with the optimal value of 'ccp_alpha'
model = DecisionTreeClassifier(ccp_alpha=0.01435396, random_state=1)
model.fit(X, y)
```

DecisionTreeClassifier(ccp\_alpha=0.01435396, random\_state=1)

Now we can tune the decision threshold probability to balance recall with another performance metrics as shown earlier in [Section 4.3](#).

## 6 Bagging

*Read section 8.2.1 of the book before using these notes.*

*Note that in this course, lecture notes are not sufficient, you must read the book for better understanding. Lecture notes are just implementing the concepts of the book on a dataset, but not explaining the concepts elaborately.*

```
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.model_selection import cross_val_score, train_test_split, KFold, GridSearchCV,
RandomizedSearchCV
from sklearn.tree import DecisionTreeRegressor, DecisionTreeClassifier
from sklearn.ensemble import BaggingRegressor, BaggingClassifier
from sklearn.linear_model import LinearRegression, LogisticRegression
from sklearn.neighbors import KNeighborsRegressor
from sklearn.metrics import roc_curve, precision_recall_curve, auc, make_scorer, recall_score,
accuracy_score, precision_score, confusion_matrix, mean_squared_error, r2_score, mean_squared_error
from skopt import BayesSearchCV
from skopt.space import Real, Integer, Categorical
from skopt.plots import plot_convergence, plot_histogram, plot_objective
from IPython import display
import itertools as it
from sklearn.preprocessing import StandardScaler

#Libraries for visualizing trees
from sklearn.tree import export_graphviz, export_text
from six import StringIO
from IPython.display import Image
import pydotplus
import time as time
import warnings
```

```
#Using the same datasets as in linear regression in STAT303-2,
#so that we can compare the non-linear models with linear regression
```

```

trainf = pd.read_csv('./Datasets/Car_features_train.csv')
trainp = pd.read_csv('./Datasets/Car_prices_train.csv')
testf = pd.read_csv('./Datasets/Car_features_test.csv')
testp = pd.read_csv('./Datasets/Car_prices_test.csv')
train = pd.merge(trainf,trainp)
test = pd.merge(testf,testp)
train.head()

```

	carID	brand	model	year	transmission	mileage	fuelType	tax	mpg	engineSize	price
0	18473	bmw	6 Series	2020	Semi-Auto	11	Diesel	145	53.3282	3.0	37980
1	15064	bmw	6 Series	2019	Semi-Auto	10813	Diesel	145	53.0430	3.0	33980
2	18268	bmw	6 Series	2020	Semi-Auto	6	Diesel	145	53.4379	3.0	36850
3	18480	bmw	6 Series	2017	Semi-Auto	18895	Diesel	145	51.5140	3.0	25998
4	18492	bmw	6 Series	2015	Automatic	62953	Diesel	160	51.4903	3.0	18990

```

X = train[['mileage','mpg','year','engineSize']]
Xtest = test[['mileage','mpg','year','engineSize']]
y = train['price']
ytest = test['price']

```

## 6.1 Bagging regression trees

Bag regression trees to develop a model to predict car price using the predictors mileage,mpg,year,and engineSize.

```

#Bagging the results of 10 decision trees to predict car price
model = BaggingRegressor(estimator=DecisionTreeRegressor(), n_estimators=10, random_state=1,
                          n_jobs=-1).fit(X, y)

```

```
np.sqrt(mean_squared_error(test.price, model.predict(Xtest)))
```

5752.0779571060875

The RMSE has reduced a lot by averaging the predictions of 10 trees. The RMSE for a single tree model with optimized parameters was around 7000.

### 6.1.1 Model accuracy vs number of trees

How does the model accuracy vary with the number of trees?

As we increase the number of trees, it will tend to reduce the variance of individual trees leading to a more accurate prediction.

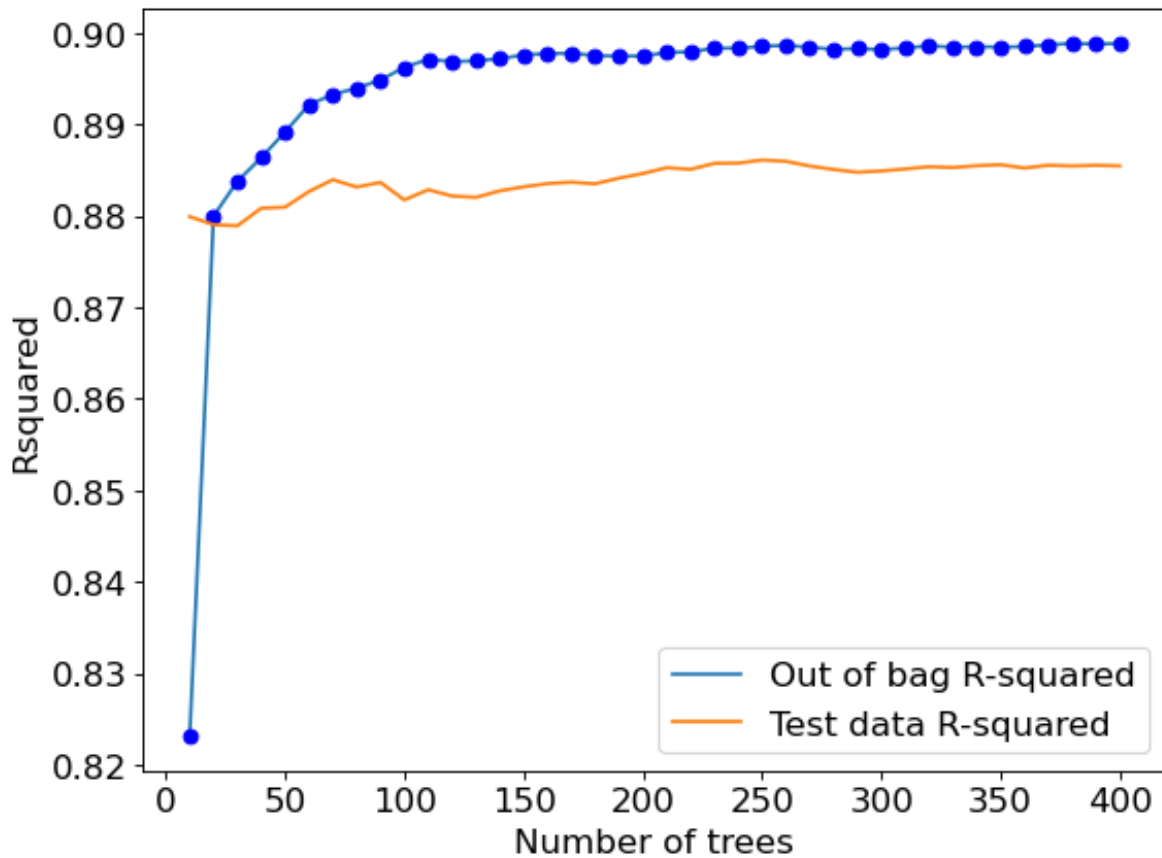
```
#Finding model accuracy vs number of trees
warnings.filterwarnings("ignore")
oob_rsquared={};test_rsquared={};oob_rmse={};test_rmse = {}
for i in np.linspace(10,400,40, dtype=int):
    model = BaggingRegressor(estimator=DecisionTreeRegressor(), n_estimators=i, random_state=
                           n_jobs=-1,oob_score=True).fit(X, y)
    oob_rsquared[i]=model.oob_score_ #Returns the out-of_bag R-squared of the model
    test_rsquared[i]=model.score(Xtest,ytest) #Returns the test R-squared of the model
    oob_rmse[i]=np.sqrt(mean_squared_error(model.oob_prediction_,y))
    test_rmse[i]=np.sqrt(mean_squared_error(model.predict(Xtest),ytest))
warnings.resetwarnings()

# The hidden warning is: "Some inputs do not have OOB scores. This probably means too few
# estimators were used to compute any reliable oob estimates." This warning will appear
# in case of small number of estimators. In such a case, some observations may be use
# by all the estimators, and their OOB score can't be computed
```

As we are bagging only 10 trees in the first iteration, some of the observations are selected in every bootstrapped sample, and thus they don't have an out-of-bag error, which is producing the warning. For every observation to have an out-of-bag error, the number of trees must be sufficiently large.

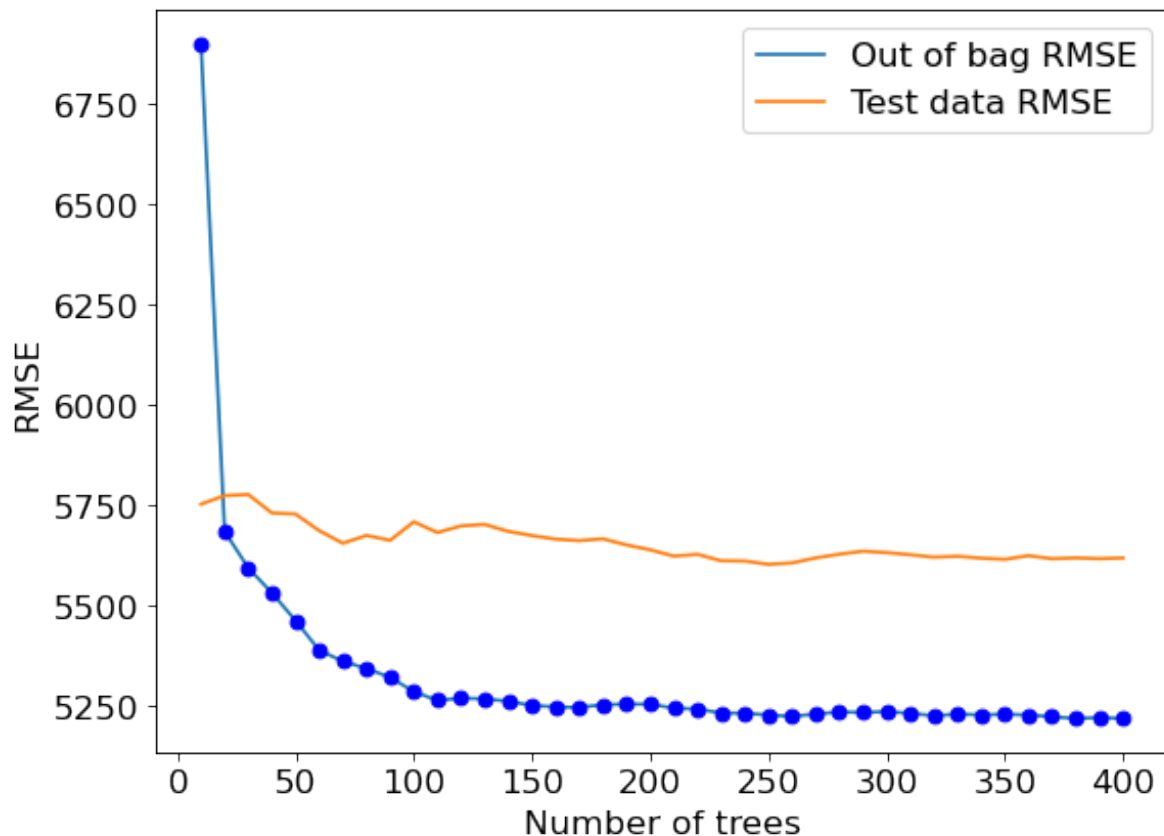
Let us visualize the out-of-bag (OOB) R-squared and R-squared on test data vs the number of trees.

```
plt.rcParams.update({'font.size': 15})
plt.figure(figsize=(8, 6), dpi=80)
plt.plot(oob_rsquared.keys(),oob_rsquared.values(),label = 'Out of bag R-squared')
plt.plot(oob_rsquared.keys(),oob_rsquared.values(),'o',color = 'blue')
plt.plot(test_rsquared.keys(),test_rsquared.values(), label = 'Test data R-squared')
plt.xlabel('Number of trees')
plt.ylabel('Rsquared')
plt.legend();
```



The out-of-bag R-squared initially increases, and then stabilizes after a certain number of trees (around 150 in this case). Note that increasing the number of trees further will not lead to overfitting. However, increasing the number of trees will increase the computations. Thus, we don't need to develop more trees once the R-squared stabilizes.

```
#Visualizing out-of-bag RMSE and test data RMSE
plt.rcParams.update({'font.size': 15})
plt.figure(figsize=(8, 6), dpi=80)
plt.plot(oob_rmse.keys(),oob_rmse.values(),label = 'Out of bag RMSE')
plt.plot(oob_rmse.keys(),oob_rmse.values(),'o',color = 'blue')
plt.plot(test_rmse.keys(),test_rmse.values(), label = 'Test data RMSE')
plt.xlabel('Number of trees')
plt.ylabel('RMSE')
plt.legend()
```



A similar trend can be seen by plotting out-of-bag RMSE and test RMSE. Note that RMSE is proportional to R-squared. We only need to visualize one of RMSE or R-squared to find the optimal number of trees.

```
#Bagging with 150 trees
model = BaggingRegressor(base_estimator=DecisionTreeRegressor(), n_estimators=150, random_state=42,
                          oob_score=True, n_jobs=-1).fit(X, y)
```

```
#OOB R-squared
model.oob_score_
```

```
0.897561533100511
```

```
#RMSE on test data
pred = model.predict(Xtest)
np.sqrt(mean_squared_error(test.price, pred))
```

```
5673.756466489405
```

### 6.1.2 Optimizing bagging hyperparameters using grid search

More parameters of a bagged regression tree model can be optimized using the typical approach of k-fold cross validation over a grid of parameter values.

Note that we don't need to tune the number of trees in bagging as we know that the higher the number of trees, the lower will be the expected MSE. So, we will tune all the hyperparameters for a fixed number of trees. Once we have obtained the optimal hyperparameter values, we'll keep increasing the number of trees until the gains are negligible.

```
n_samples = train.shape[0]
n_features = train.shape[1]

params = {'base_estimator': [DecisionTreeRegressor(random_state = 1), LinearRegression()], #Cor
        'n_estimators': [100],
        'max_samples': [0.5, 1.0],
        'max_features': [0.5, 1.0],
        'bootstrap': [True, False],
        'bootstrap_features': [True, False]}

cv = KFold(n_splits=5, shuffle=True, random_state=1)
bagging_regressor_grid = GridSearchCV(BaggingRegressor(random_state=1, n_jobs=-1),
                                     param_grid=params, cv=cv, n_jobs=-1, verbose=1)
bagging_regressor_grid.fit(X, y)

print('Train R^2 Score : %.3f'%bagging_regressor_grid.best_estimator_.score(X, y))
print('Test R^2 Score : %.3f'%bagging_regressor_grid.best_estimator_.score(Xtest, ytest))
print('Best R^2 Score Through Grid Search : %.3f'%bagging_regressor_grid.best_score_)
print('Best Parameters : ', bagging_regressor_grid.best_params_)
```

Fitting 5 folds for each of 32 candidates, totalling 160 fits

Train R^2 Score : 0.986

Test R^2 Score : 0.882

Best R^2 Score Through Grid Search : 0.892

Best Parameters : {'base\_estimator': DecisionTreeRegressor(random\_state=1), 'bootstrap': Tr

You may use the object `bagging_regressor_grid` to directly make the prediction.

```
np.sqrt(mean_squared_error(test.price, bagging_regressor_grid.predict(Xtest)))
```

5708.308794847089



Note that once the model has been tuned and the optimal hyperparameters identified, we can keep increasing the number of trees until it ceases to benefit.

```
#Model with optimal hyperparameters and increased number of trees
model = BaggingRegressor(base_estimator=DecisionTreeRegressor(), n_estimators=500, random_state=0,
                        oob_score=True, n_jobs=-1, bootstrap_features=False, bootstrap=True,
                        max_features=1.0, max_samples=1.0).fit(X, y)
```

```
#RMSE on test data
np.sqrt(mean_squared_error(test.price, model.predict(Xtest)))
```

5624.685464926517

## 6.2 Bagging for classification

Bag classification tree models to predict if a person has diabetes.

```
train = pd.read_csv('./Datasets/diabetes_train.csv')
test = pd.read_csv('./Datasets/diabetes_test.csv')
```

```
X = train.drop(columns = 'Outcome')
Xtest = test.drop(columns = 'Outcome')
y = train['Outcome']
ytest = test['Outcome']
```

```
#Bagging the results of 10 decision trees to predict car price
model = BaggingClassifier(base_estimator=DecisionTreeClassifier(), n_estimators=150, random_state=0,
                        n_jobs=-1).fit(X, y)
```

```
# Performance metrics computation for the optimum decision threshold probability
desired_threshold = 0.23
```

```
y_pred_prob = model.predict_proba(Xtest)[:,-1]
```

```
# Classifying observations in the positive class (y = 1) if the predicted probability is greater
# than the desired decision threshold probability
y_pred = y_pred_prob > desired_threshold
y_pred = y_pred.astype(int)
```

```

#Computing the accuracy
print("Accuracy: ",accuracy_score(y_pred, ytest)*100)

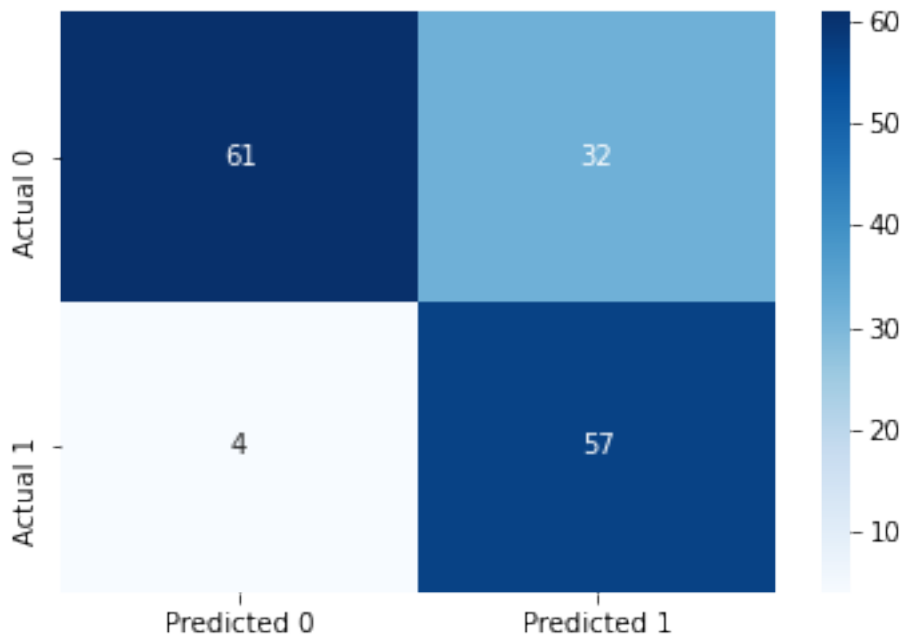
#Computing the ROC-AUC
fpr, tpr, auc_thresholds = roc_curve(ytest, y_pred_prob)
print("ROC-AUC: ",auc(fpr, tpr))# AUC of ROC

#Computing the precision and recall
print("Precision: ", precision_score(ytest, y_pred))
print("Recall: ", recall_score(ytest, y_pred))

#Confusion matrix
cm = pd.DataFrame(confusion_matrix(ytest, y_pred),
                  columns=['Predicted 0', 'Predicted 1'], index = ['Actual 0', 'Actual 1'])
sns.heatmap(cm, annot=True, cmap='Blues', fmt='g');

```

Accuracy: 76.62337662337663  
 ROC-AUC: 0.8766084963863917  
 Precision: 0.6404494382022472  
 Recall: 0.9344262295081968



As a result of bagging, we obtain a model (with a threshold probability cutoff of 0.23) that has a better performance on test data in terms of almost all the metrics - accuracy, precision

(comparable performance), recall, and ROC-AUC, as compared the single tree classification model (with a threshold probability cutoff of 0.23). Note that we have not yet tuned the model using `GridSearchCv` here, which is shown towards the end of this chapter.

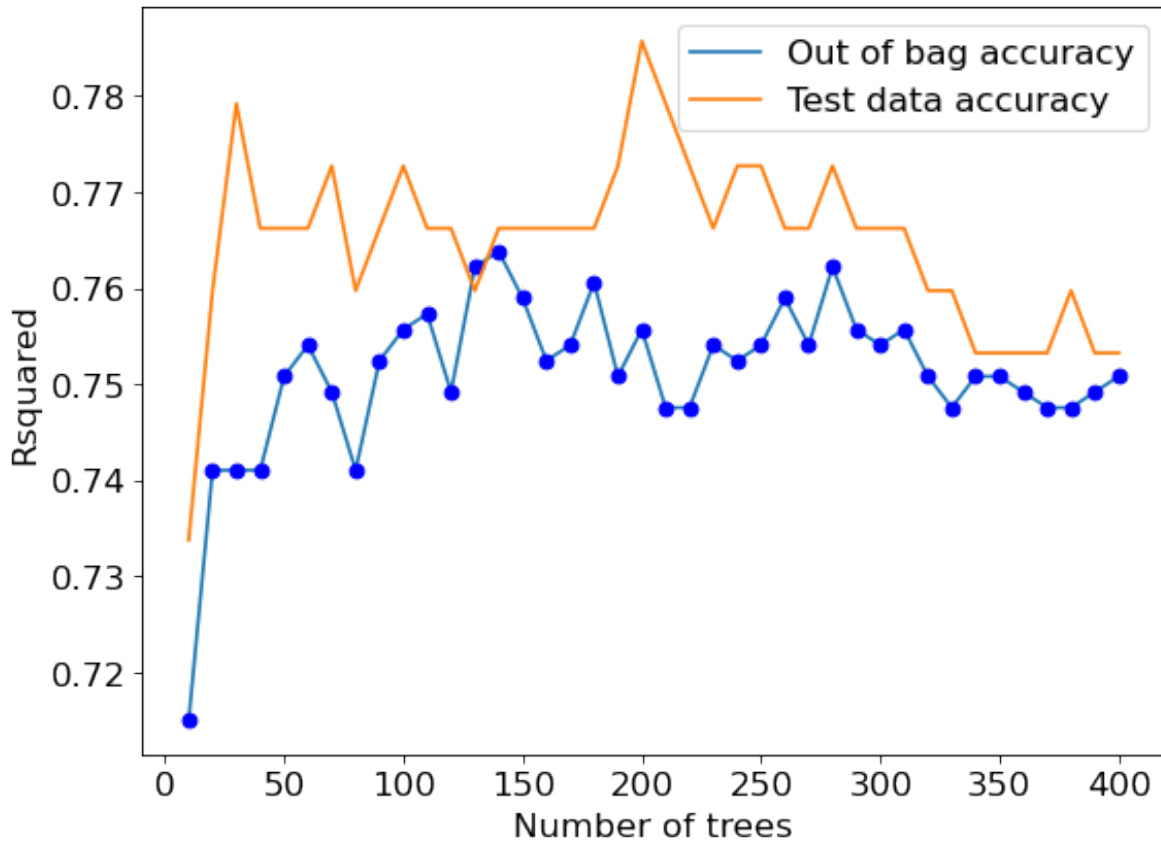
### 6.2.1 Model accuracy vs number of trees

```
#Finding model accuracy vs number of trees
oob_accuracy={};test_accuracy={};oob_rmse={};test_rmse = {}
for i in np.linspace(10,400,40,dtype=int):
    model = BaggingClassifier(base_estimator=DecisionTreeClassifier(), n_estimators=i, random
                             n_jobs=-1,oob_score=True).fit(X, y)
    oob_accuracy[i]=model.oob_score_ #Returns the out-of_bag R-squared of the model
    test_accuracy[i]=model.score(Xtest,ytest) #Returns the test R-squared of the model
```

```
C:\Users\akl0407\Anaconda3\lib\site-packages\sklearn\ensemble\_bagging.py:640: UserWarning: 
warn("Some inputs do not have OOB scores. "
```

```
C:\Users\akl0407\Anaconda3\lib\site-packages\sklearn\ensemble\_bagging.py:644: RuntimeWarning
oob_decision_function = (predictions /
```

```
plt.rcParams.update({'font.size': 15})
plt.figure(figsize=(8, 6), dpi=80)
plt.plot(oob_accuracy.keys(),oob_accuracy.values(),label = 'Out of bag accuracy')
plt.plot(oob_accuracy.keys(),oob_accuracy.values(),'o',color = 'blue')
plt.plot(test_accuracy.keys(),test_accuracy.values(), label = 'Test data accuracy')
plt.xlabel('Number of trees')
plt.ylabel('Rsquared')
plt.legend()
```

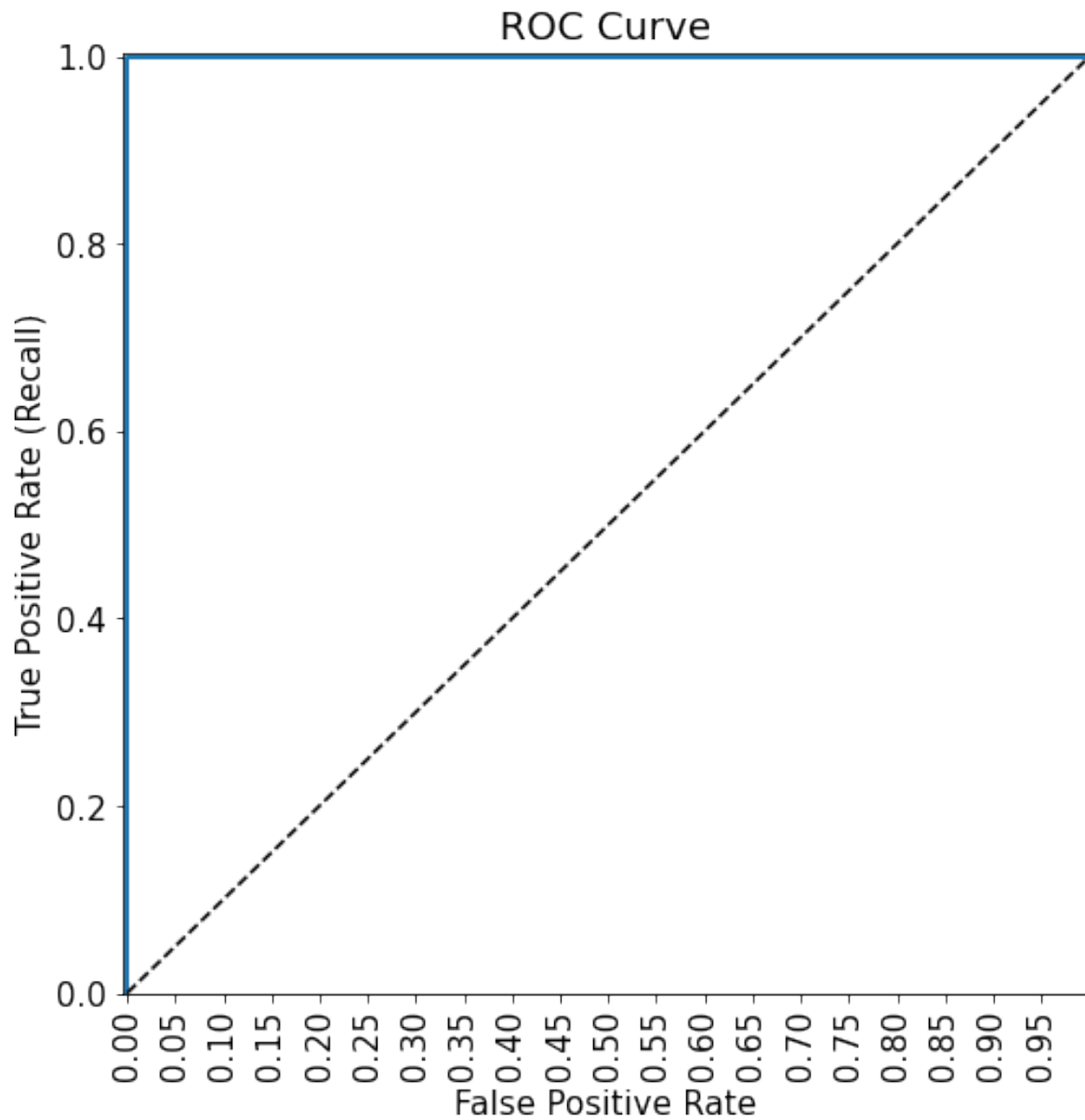


```
#ROC curve on training data
ypred = model.predict_proba(X)[: , 1]
fpr, tpr, auc_thresholds = roc_curve(y, ypred)
print(auc(fpr, tpr))# AUC of ROC
def plot_roc_curve(fpr, tpr, label=None):

    plt.figure(figsize=(8,8))
    plt.title('ROC Curve')
    plt.plot(fpr, tpr, linewidth=2, label=label)
    plt.plot([0, 1], [0, 1], 'k--')
    plt.axis([-0.005, 1, 0, 1.005])
    plt.xticks(np.arange(0,1, 0.05), rotation=90)
    plt.xlabel("False Positive Rate")
    plt.ylabel("True Positive Rate (Recall)")

fpr, tpr, auc_thresholds = roc_curve(y, ypred)
plot_roc_curve(fpr, tpr)
```

1.0



Note that there is perfect separation in train data as  $\text{ROC-AUC} = 1$ . This shows that the model is probably overfitting. However, this also shows that, despite the reduced variance (*as compared to a single tree*), the bagged tree model is flexibly enough to perfectly separate the classes.

```

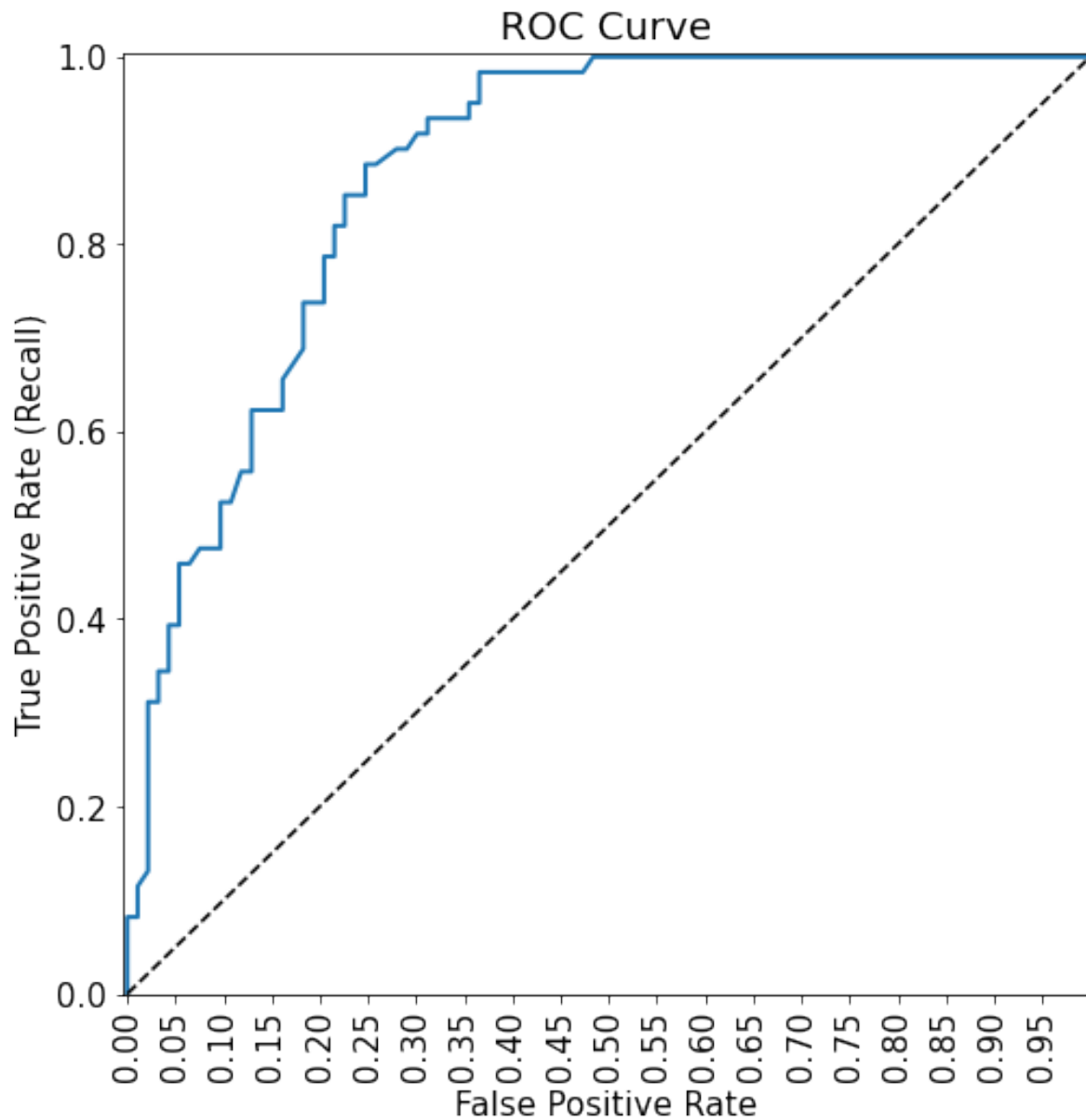
#ROC curve on test data
ypred = model.predict_proba(Xtest)[: , 1]
fpr, tpr, auc_thresholds = roc_curve(ytest, ypred)
print("ROC-AUC = ",auc(fpr, tpr))# AUC of ROC
def plot_roc_curve(fpr, tpr, label=None):

    plt.figure(figsize=(8,8))
    plt.title('ROC Curve')
    plt.plot(fpr, tpr, linewidth=2, label=label)
    plt.plot([0, 1], [0, 1], 'k--')
    plt.axis([-0.005, 1, 0, 1.005])
    plt.xticks(np.arange(0,1, 0.05), rotation=90)
    plt.xlabel("False Positive Rate")
    plt.ylabel("True Positive Rate (Recall)")

fpr, tpr, auc_thresholds = roc_curve(ytest, ypred)
plot_roc_curve(fpr, tpr)

```

ROC-AUC = 0.8781949585757096



### 6.2.2 Optimizing bagging hyperparameters using grid search

More parameters of a bagged classification tree model can be optimized using the typical approach of k-fold cross validation over a grid of parameter values.

```

n_samples = train.shape[0]
n_features = train.shape[1]

params = {'base_estimator': [DecisionTreeClassifier(random_state = 1), LogisticRegression()],
          'n_estimators': [150, 200, 250],
          'max_samples': [0.5, 1.0],
          'max_features': [0.5, 1.0],
          'bootstrap': [True, False],
          'bootstrap_features': [True, False]}

cv = KFold(n_splits=5, shuffle=True, random_state=1)
bagging_classifier_grid = GridSearchCV(BaggingClassifier(random_state=1, n_jobs=-1),
                                       param_grid=params, cv=cv, n_jobs=-1, verbose=1,
                                       scoring = ['precision', 'recall'], refit='recall')
bagging_classifier_grid.fit(X, y)

print('Train accuracy : %.3f'%bagging_classifier_grid.best_estimator_.score(X, y))
print('Test accuracy : %.3f'%bagging_classifier_grid.best_estimator_.score(Xtest, ytest))
print('Best accuracy Through Grid Search : %.3f'%bagging_classifier_grid.best_score_)
print('Best Parameters : ', bagging_classifier_grid.best_params_)

```

Fitting 5 folds for each of 96 candidates, totalling 480 fits

Train accuracy : 1.000

Test accuracy : 0.786

Best accuracy Through Grid Search : 0.573

Best Parameters : {'base\_estimator': DecisionTreeClassifier(random\_state=1), 'bootstrap': True}

### 6.2.3 Tuning the decision threshold probability

We'll find a decision threshold probability that balances recall with precision.

```

model = BaggingClassifier(base_estimator=DecisionTreeClassifier(random_state=1), n_estimators=
                          random_state=1, max_features=1.0, oob_score=True,
                          max_samples=1.0, n_jobs=-1, bootstrap=True, bootstrap_features=False).fit(X, y)

```

As the model is overfitting on the train data, it will not be a good idea to tune the decision threshold probability based on the precision-recall curve on train data, as shown in the figure below.

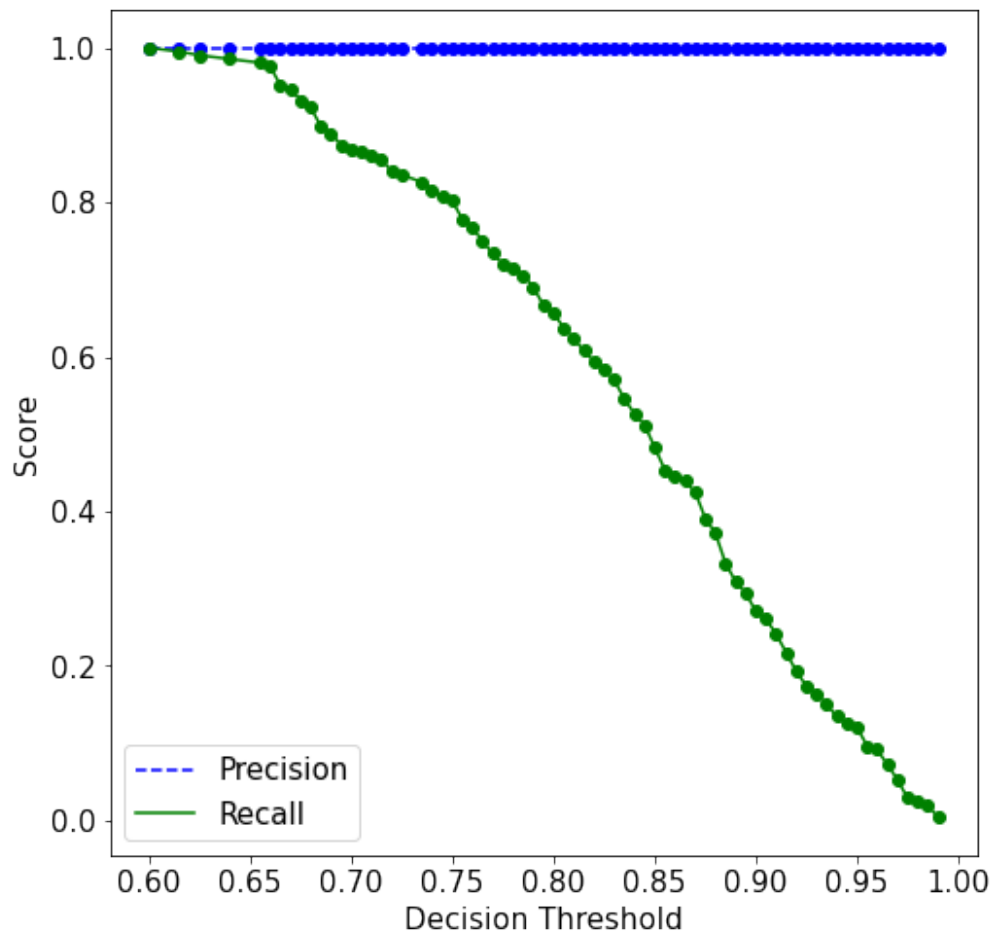


```

ypred = model.predict_proba(X)[:,-1]
p, r, thresholds = precision_recall_curve(y, ypred)
def plot_precision_recall_vs_threshold(precisions, recalls, thresholds):
    plt.figure(figsize=(8, 8))
    plt.title("Precision and Recall Scores as a function of the decision threshold")
    plt.plot(thresholds, precisions[:-1], "b--", label="Precision")
    plt.plot(thresholds, recalls[:-1], "g-", label="Recall")
    plt.plot(thresholds, precisions[:-1], "o", color = 'blue')
    plt.plot(thresholds, recalls[:-1], "o", color = 'green')
    plt.ylabel("Score")
    plt.xlabel("Decision Threshold")
    plt.legend(loc='best')
    plt.legend()
plot_precision_recall_vs_threshold(p, r, thresholds)

```

Precision and Recall Scores as a function of the decision threshold

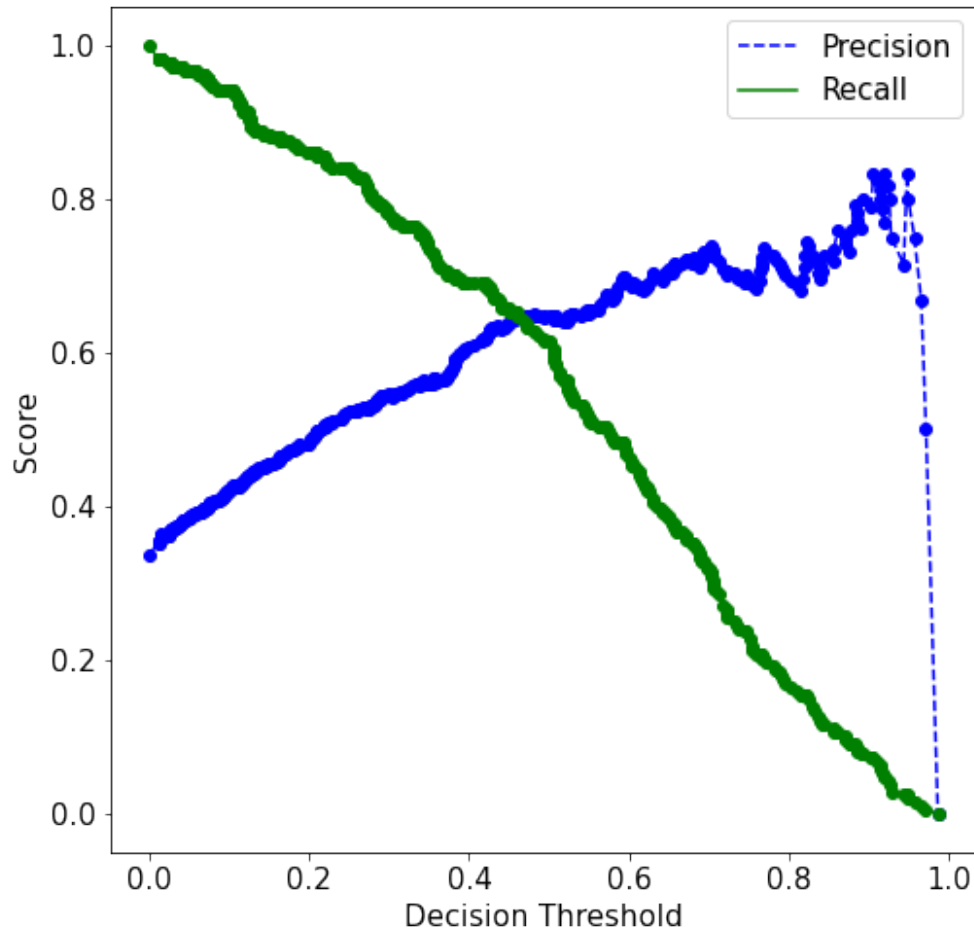


Instead, we should make the precision-recall curve using the out-of-bag predictions, as shown below. The method `oob_decision_function_` provides the predicted probability.

```
ypred = model.oob_decision_function_[:,1]
p, r, thresholds = precision_recall_curve(y, ypred)
def plot_precision_recall_vs_threshold(precisions, recalls, thresholds):
    plt.figure(figsize=(8, 8))
    plt.title("Precision and Recall Scores as a function of the decision threshold")
    plt.plot(thresholds, precisions[:-1], "b--", label="Precision")
    plt.plot(thresholds, recalls[:-1], "g-", label="Recall")
    plt.plot(thresholds, precisions[:-1], "o", color = 'blue')
    plt.plot(thresholds, recalls[:-1], "o", color = 'green')
    plt.ylabel("Score")
    plt.xlabel("Decision Threshold")
```

```
plt.legend(loc='best')
plt.legend()
plot_precision_recall_vs_threshold(p, r, thresholds)
```

Precision and Recall Scores as a function of the decision threshold



```
# Thresholds with precision and recall
all_thresholds = np.concatenate([thresholds.reshape(-1,1), p[:-1].reshape(-1,1), r[:-1].reshape(-1,1)])
recall_more_than_80 = all_thresholds[all_thresholds[:,2]>0.8,:]
# As the values in 'recall_more_than_80' are arranged in decreasing order of recall and increasing order of precision,
# the last value will provide the maximum threshold probability for the recall to be more than 0.8
# We wish to find the maximum threshold probability to obtain the maximum possible precision
recall_more_than_80[recall_more_than_80.shape[0]-1]
```

```
array([0.2804878 , 0.53205128, 0.80193237])
```

Suppose, we wish to have at least 80% recall, with the highest possible precision. Then, based on the precision-recall curve, we should have a decision threshold probability of 0.28.

```
# Performance metrics computation for the optimum decision threshold probability
desired_threshold = 0.28

y_pred_prob = model.predict_proba(Xtest)[:,-1]

# Classifying observations in the positive class (y = 1) if the predicted probability is greater
# than the desired decision threshold probability
y_pred = y_pred_prob > desired_threshold
y_pred = y_pred.astype(int)

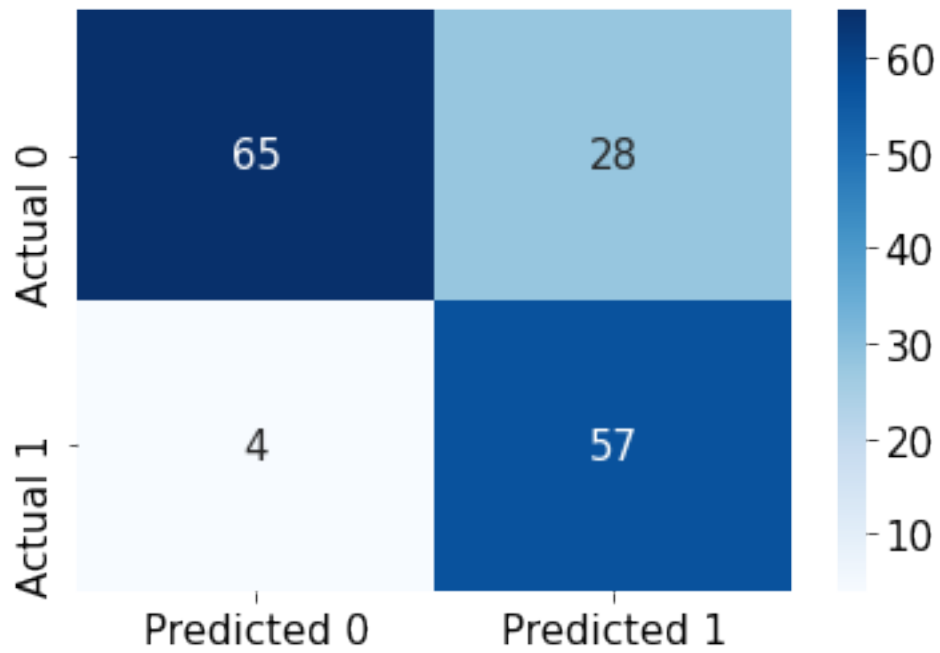
#Computing the accuracy
print("Accuracy: ",accuracy_score(y_pred, ytest)*100)

#Computing the ROC-AUC
fpr, tpr, auc_thresholds = roc_curve(ytest, y_pred_prob)
print("ROC-AUC: ",auc(fpr, tpr))# AUC of ROC

#Computing the precision and recall
print("Precision: ", precision_score(ytest, y_pred))
print("Recall: ", recall_score(ytest, y_pred))

#Confusion matrix
cm = pd.DataFrame(confusion_matrix(ytest, y_pred),
                  columns=['Predicted 0', 'Predicted 1'], index = ['Actual 0', 'Actual 1'])
sns.heatmap(cm, annot=True, cmap='Blues', fmt='g');
```

```
Accuracy: 79.22077922077922
ROC-AUC: 0.8802221047065044
Precision: 0.6705882352941176
Recall: 0.9344262295081968
```



Note that this model has a better performance than the untuned bagged model earlier, and the single tree classification model, as expected.

## 7 Bagging (addendum)

This notebook provides examples to:

1. Compare tuning bagging hyperparameters with OOB validation and  $k$ -fold cross-validation.
2. Compare bagging tuned models with untuned models.

```
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.model_selection import cross_val_score, train_test_split, KFold, GridSearchCV, Pa
RandomizedSearchCV, RepeatedKFold
from sklearn.tree import DecisionTreeRegressor, DecisionTreeClassifier
from sklearn.ensemble import BaggingRegressor, BaggingClassifier
from sklearn.linear_model import LinearRegression, LogisticRegression
from sklearn.neighbors import KNeighborsRegressor
from sklearn.metrics import roc_curve, precision_recall_curve, auc, make_scorer, recall_score,
accuracy_score, precision_score, confusion_matrix, mean_squared_error, r2_score, mean_squared
from skopt import BayesSearchCV
from skopt.space import Real, Integer, Categorical
from skopt.plots import plot_convergence, plot_histogram, plot_objective
from IPython import display
import itertools as it

#Libraries for visualizing trees
from sklearn.tree import export_graphviz, export_text
from six import StringIO
from IPython.display import Image
import pydotplus
import time as time
import warnings
```

```
#Using the same datasets as in linear regression in STAT303-2,
#so that we can compare the non-linear models with linear regression
trainf = pd.read_csv('./Datasets/Car_features_train.csv')
trainp = pd.read_csv('./Datasets/Car_prices_train.csv')
testf = pd.read_csv('./Datasets/Car_features_test.csv')
testp = pd.read_csv('./Datasets/Car_prices_test.csv')
train = pd.merge(trainf,trainp)
test = pd.merge(testf,testp)
train.head()
```

	carID	brand	model	year	transmission	mileage	fuelType	tax	mpg	engineSize	price
0	18473	bmw	6 Series	2020	Semi-Auto	11	Diesel	145	53.3282	3.0	37980
1	15064	bmw	6 Series	2019	Semi-Auto	10813	Diesel	145	53.0430	3.0	33980
2	18268	bmw	6 Series	2020	Semi-Auto	6	Diesel	145	53.4379	3.0	36850
3	18480	bmw	6 Series	2017	Semi-Auto	18895	Diesel	145	51.5140	3.0	25998
4	18492	bmw	6 Series	2015	Automatic	62953	Diesel	160	51.4903	3.0	18990

```
X = train[['mileage','mpg','year','engineSize']]
Xtest = test[['mileage','mpg','year','engineSize']]
y = train['price']
ytest = test['price']
```

1. Tree without tuning
2. Tree performance improves with tuning
3. Bagging tuned tree
4. Bagging untuned tree - better, how?
5. Tuning bagged model - OOB
6. Tuning bagged model - BayesSearchCV
7. warm start
8. Bagging KNN - no need to tune number of neighbors

## 7.1 Tree without tuning

```
model = DecisionTreeRegressor()
cv = KFold(n_splits=5, shuffle=True, random_state=1)
-np.mean(cross_val_score(model, X, y, scoring='neg_root_mean_squared_error', cv = cv))
```

7056.960817154941

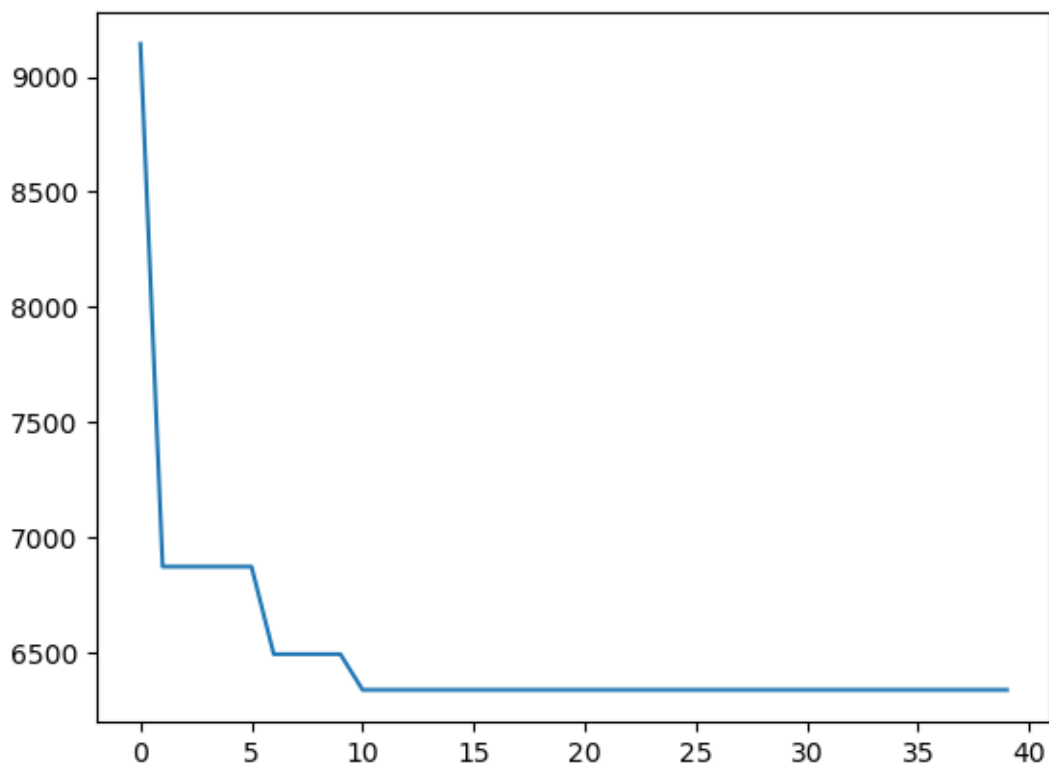
```

param_grid = {'max_depth': Integer(2, 30)}
gcv = BayesSearchCV(model, search_spaces = param_grid, cv = cv, n_iter = 40, random_state = 1,
                    scoring = 'neg_root_mean_squared_error', n_jobs = -1)
paras = list(gcv.search_spaces.keys())
paras.sort()

def monitor(optim_result):
    cv_values = pd.Series(optim_result['func_vals']).cummin()
    display.clear_output(wait = True)
    min_ind = pd.Series(optim_result['func_vals']).argmin()
    print(paras, "=", optim_result['x_iters'][min_ind], pd.Series(optim_result['func_vals']))
    sns.lineplot(cv_values)
    plt.show()
gcv.fit(X, y, callback = monitor)

```

['max\_depth'] = [10] 6341.1481858990355



BayesSearchCV(cv=KFold(n\_splits=5, random\_state=1, shuffle=True),



```
estimator=DecisionTreeRegressor(), n_iter=40, n_jobs=-1,
random_state=10, scoring='neg_root_mean_squared_error',
search_spaces={'max_depth': Integer(low=2, high=30, prior='uniform', transform
```

## 7.2 Performance of tree improves with tuning

```
model = DecisionTreeRegressor(max_depth=10)
cv = KFold(n_splits=5, shuffle=True, random_state=1)
-np.mean(cross_val_score(model, X, y, scoring='neg_root_mean_squared_error', cv = cv))
```

6442.494300778735

## 7.3 Bagging tuned trees

```
model = BaggingRegressor(DecisionTreeRegressor(max_depth = 10), oob_score=True, n_estimators
mean_squared_error(model.oob_prediction_, y, squared = False)
```

5354.357809020438

## 7.4 Bagging untuned trees

```
model = BaggingRegressor(DecisionTreeRegressor(), oob_score=True, n_estimators = 100).fit(X,
mean_squared_error(model.oob_prediction_, y, squared = False)
```

5248.720845665685

### Why is bagging tuned trees worse than bagging untuned trees?

In the tuned tree here, the reduction in variance by controlling maximum depth resulted in an increase in bias of individual trees. Bagging trees only reduces the variance, but not the bias of the individual trees. Thus, bagging high bias models will result in a high-bias model, while bagging high variance models may result in a low variance model if the models are not highly correlated.

Bagging tuned models may provide a better performance as compared to bagging untuned models if the reduction in variance of the individual models is high enough to overshadow the increase in bias, and increase in pairwise correlation of the individual models.

## 7.5 Tuning bagged model - OOB

```
param_grid1 = {'max_samples': [0.25, 0.5, 0.75, 1.0],
               'max_features': [2, 3, 4],
               'bootstrap_features': [True, False]}
param_grid2 = {'max_samples': [0.25, 0.5, 0.75, 1.0],
               'max_features': [1],
               'bootstrap_features': [False]}
param_list1 = list(it.product(*[values for key, values in param_grid1.items()]))
param_list2 = list(it.product(*[values for key, values in param_grid2.items()]))
param_list = param_list1 + param_list2

oob_score_pr = []
for pr in param_list:
    model = BaggingRegressor(DecisionTreeRegressor(), max_samples=pr[0], max_features=pr[1],
                             bootstrap_features=pr[2], n_jobs = -1, oob_score=True, n_estimators=100)
    oob_score_pr.append(mean_squared_error(model.oob_prediction_, y, squared=False))
```

### What is the benefit of OOB validation to tune hyperparameters in bagging?

It is much cheaper than  $k$ -fold cross-validation, as only  $1/k$  of the models are trained with OOB validation as compared to  $k$ -fold cross-validation. However, the cost of training individual models is lower in  $k$ -fold cross-validation as models are trained on a smaller dataset. Typically, OOB will be faster than  $k$ -fold cross-validation. The higher the value of  $k$ , the more faster OOB validation will be as compared to  $k$ -fold cross-validation.

## 7.6 Tuning without k-fold cross-validation

When hyperparameters can be tuned with OOB validation, what is the benefit of using  $k$ -fold cross-validation?

1. Hyperparameters cannot be tuned over continuous spaces with OOB validation.
2. OOB score is not computed if sampling is done without replacement (*bootstrap = False*). Thus, for tuning the *bootstrap* hyperparameter,  $k$ -fold cross-validation will need to be used.

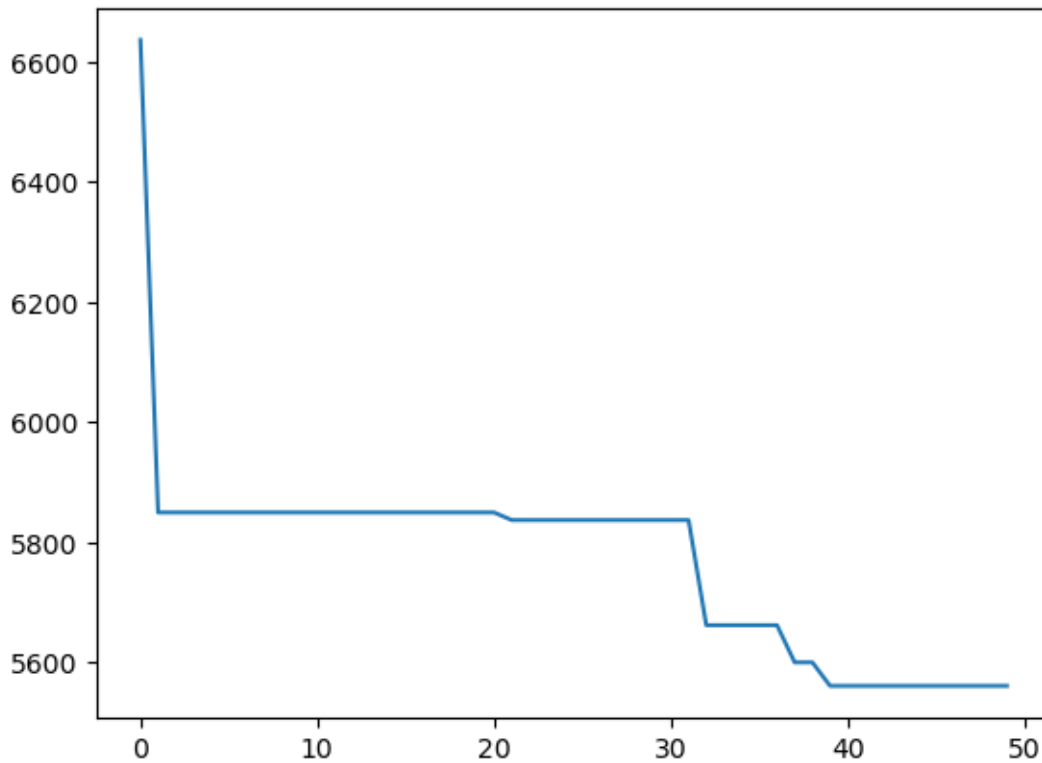
```
def monitor(optim_result):
    cv_values = pd.Series(optim_result['func_vals']).cummin()
    display.clear_output(wait = True)
    min_ind = pd.Series(optim_result['func_vals']).argmin()
    print(paras, "=", optim_result['x_iters'][min_ind], pd.Series(optim_result['func_vals']))
    sns.lineplot(cv_values)
    plt.show()
```

```
param_grid = {'max_samples': Real(0.2, 1.0),
              'max_features': Integer(1, 4),
              'bootstrap_features': [True, False],
              'bootstrap': [True, False]}
gcv = BayesSearchCV(BaggingRegressor(DecisionTreeRegressor(), bootstrap=False),
                    search_spaces = param_grid, cv = cv, n_jobs = -1,
                    scoring='neg_root_mean_squared_error')

paras = list(gcv.search_spaces.keys())
paras.sort()

gcv.fit(X, y, callback=monitor)
```

```
['bootstrap', 'bootstrap_features', 'max_features', 'max_samples'] = [True, False, 4, 0.8061]
```

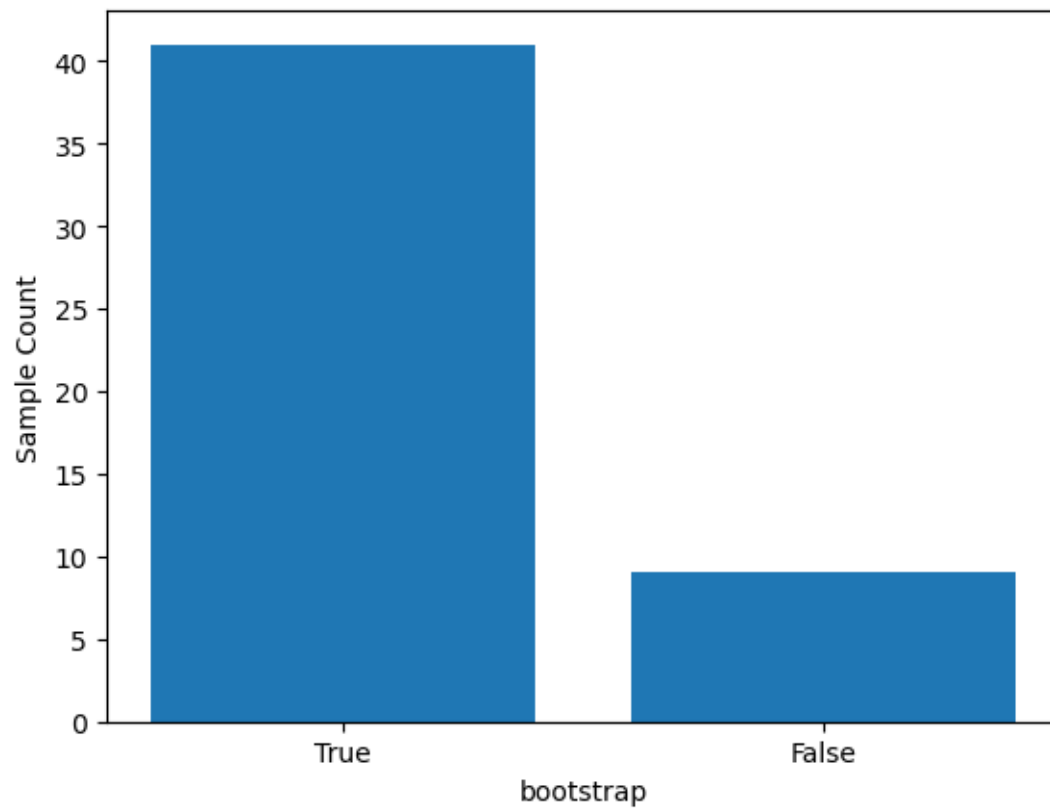


```

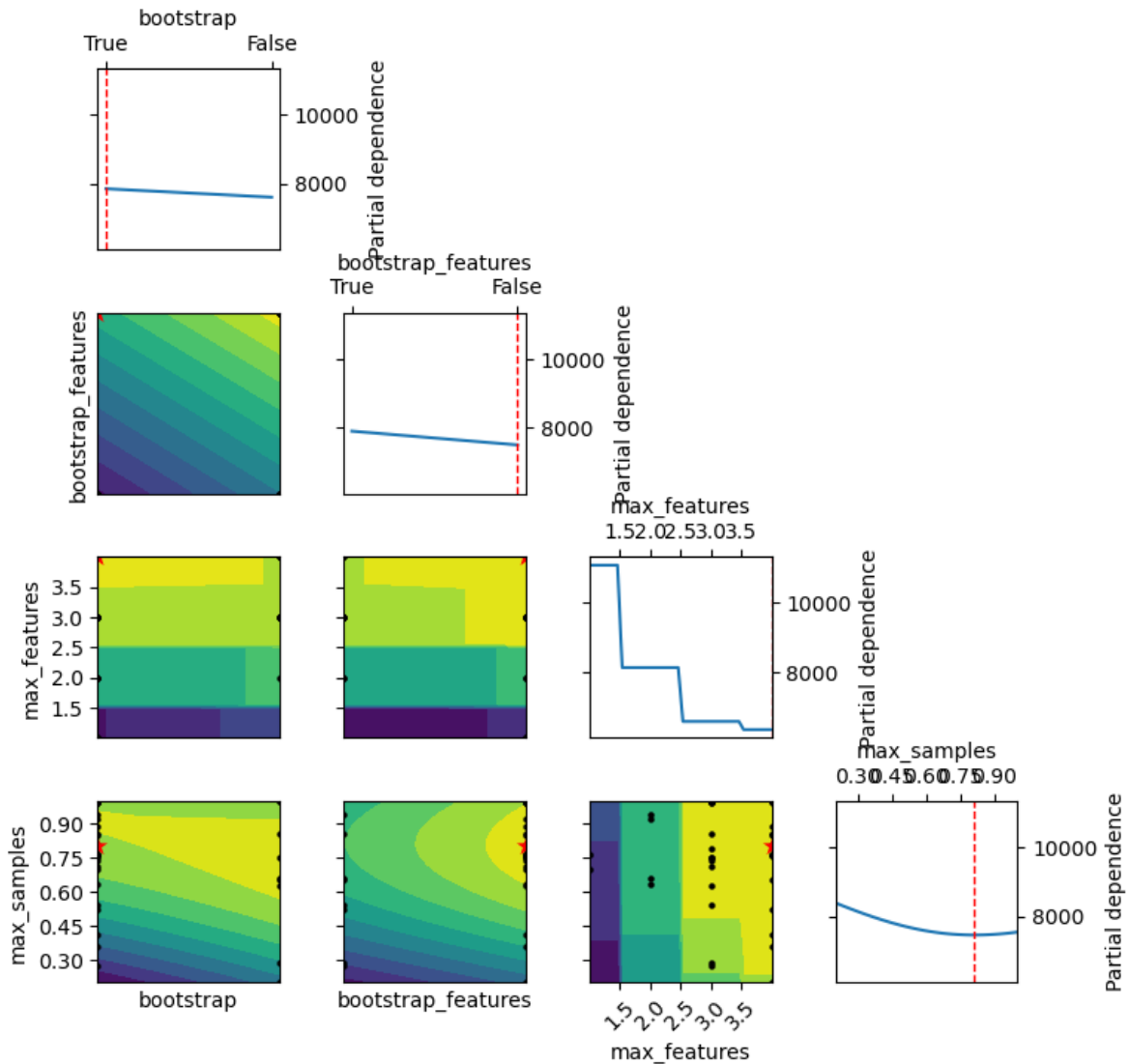
BayesSearchCV(cv=KFold(n_splits=5, random_state=1, shuffle=True),
               estimator=BaggingRegressor(bootstrap=False,
                                           estimator=DecisionTreeRegressor()),
               n_jobs=-1, scoring='neg_root_mean_squared_error',
               search_spaces={'bootstrap': [True, False],
                              'bootstrap_features': [True, False],
                              'max_features': Integer(low=1, high=4, prior='uniform', transform=None),
                              'max_samples': Real(low=0.2, high=1.0, prior='uniform', transform=None)},
               verbose=1)

plot_histogram(gcv.optimizer_results_[0],0)

```



```
plot_objective(gcv.optimizer_results_[0])
```



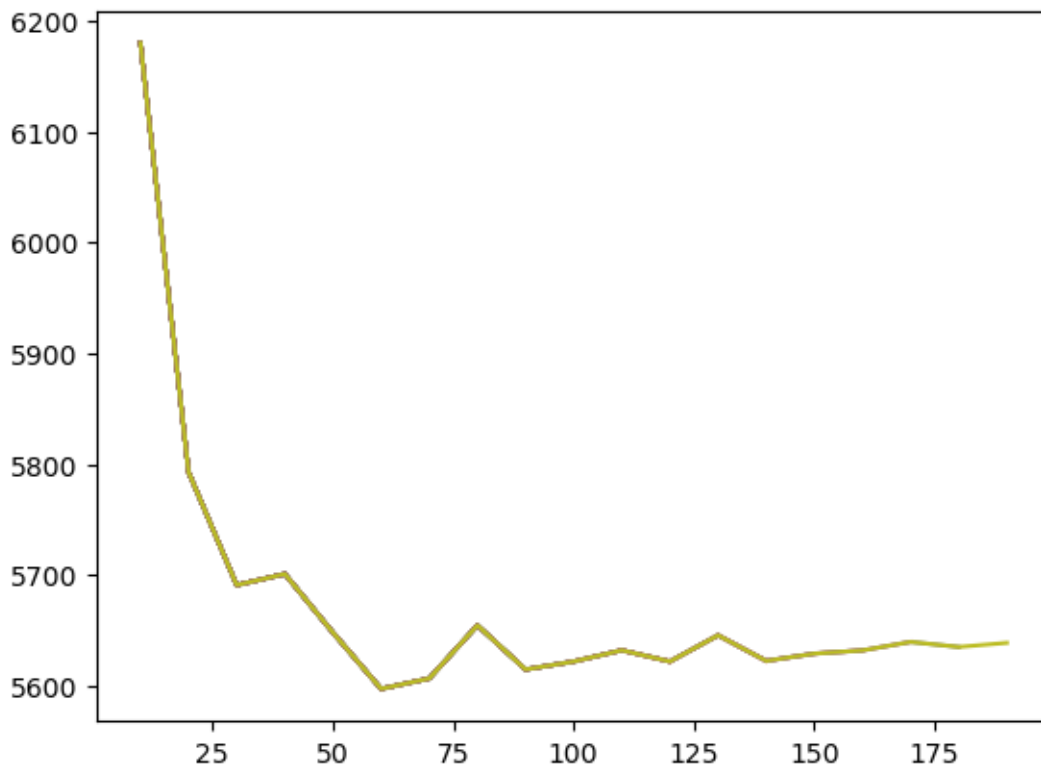
## 7.7 warm start

What is the purpose of `warm_start`?

The purpose of `warm_start` is to avoid developing trees from scratch, and incrementally add trees to monitor the validation error. However, note that OOB score is not computed with `warm_start`. Thus, a validation set approach will need to be adopted to tune number of trees.

A cheaper approach to tune number of estimators is to just use trial and error, and stop increasing once the cross-validation error / OOB error / validation set error stabilizes.

```
model = BaggingRegressor(DecisionTreeRegressor(), oob_score=False, n_estimators = 5,  
                        warm_start=True).fit(X, y)  
  
rmse = []  
for i in range(10, 200, 10):  
    model.n_estimators = i  
    model.fit(X, y)  
    rmse.append(mean_squared_error(model.predict(Xtest), ytest, squared=False))  
    sns.lineplot(x = range(10, i+1, 10), y = rmse)
```



## 7.8 Bagging KNN

Should we bag a tuned KNN model or an untuned one?

```
from sklearn.preprocessing import StandardScaler
```

```
model = KNeighborsRegressor(n_neighbors=9) # optimal neighbors
scaler = StandardScaler()
scaler.fit(X)
X_scaled = scaler.transform(X)
-np.mean(cross_val_score((model), X_scaled, y, cv = cv,
                        scoring='neg_root_mean_squared_error', n_jobs = -1))
```

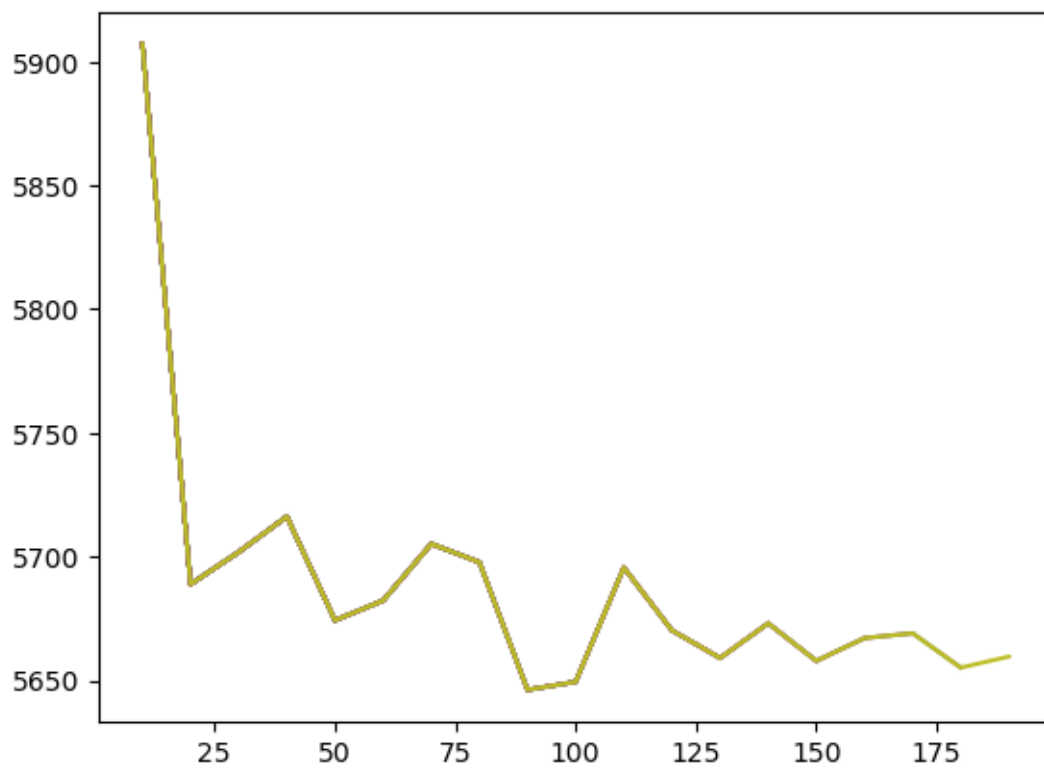
6972.997277781689

```
model = KNeighborsRegressor(n_neighbors=1)
scaler = StandardScaler()
scaler.fit(X)
X_scaled = scaler.transform(X)
-np.mean(cross_val_score(BaggingRegressor(model), X_scaled, y, cv = cv,
                        scoring='neg_root_mean_squared_error', n_jobs = -1))
```

6254.305462266355

```
model = BaggingRegressor(DecisionTreeRegressor(), n_estimators=5, warm_start=True)
model.fit(X, y)
rmse = []
for i in range(10, 200, 10):
    model.n_estimators = i
    model.fit(X, y)
    rmse.append(mean_squared_error(model.predict(Xtest), ytest, squared=False))
sns.lineplot(x = range(10, i + 1, 10), y = rmse)
```





## 8 Random Forest

*Read section 8.2.2 of the book before using these notes.*

*Note that in this course, lecture notes are not sufficient, you must read the book for better understanding. Lecture notes are just implementing the concepts of the book on a dataset, but not explaining the concepts elaborately.*

```
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import cross_val_score, train_test_split
from sklearn.model_selection import KFold
from sklearn.tree import DecisionTreeRegressor, DecisionTreeClassifier
from sklearn.model_selection import GridSearchCV, ParameterGrid
from sklearn.ensemble import BaggingRegressor, BaggingClassifier, RandomForestRegressor, RandomForestClassifier
from sklearn.linear_model import LinearRegression, LogisticRegression
from sklearn.neighbors import KNeighborsRegressor
from sklearn.metrics import roc_curve, precision_recall_curve, auc, make_scorer, recall_score, accuracy_score, precision_score, confusion_matrix, mean_squared_error, r2_score

import itertools as it

#Libraries for visualizing trees
from sklearn.tree import export_graphviz
from six import StringIO
from IPython.display import Image
import pydotplus
import time as time
import warnings

#Using the same datasets as used for linear regression in STAT303-2,
#so that we can compare the non-linear models with linear regression
trainf = pd.read_csv('./Datasets/Car_features_train.csv')
trainp = pd.read_csv('./Datasets/Car_prices_train.csv')
```

```
testf = pd.read_csv('./Datasets/Car_features_test.csv')
testp = pd.read_csv('./Datasets/Car_prices_test.csv')
train = pd.merge(trainf, trainp)
test = pd.merge(testf, testp)
train.head()
```

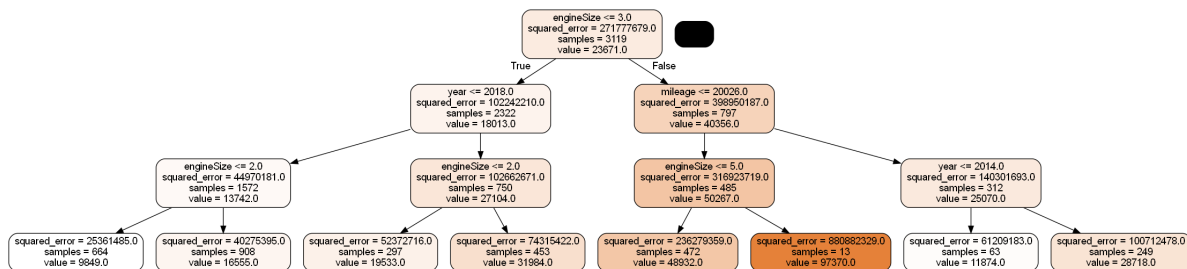
	carID	brand	model	year	transmission	mileage	fuelType	tax	mpg	engineSize	price
0	18473	bmw	6 Series	2020	Semi-Auto	11	Diesel	145	53.3282	3.0	37980
1	15064	bmw	6 Series	2019	Semi-Auto	10813	Diesel	145	53.0430	3.0	33980
2	18268	bmw	6 Series	2020	Semi-Auto	6	Diesel	145	53.4379	3.0	36850
3	18480	bmw	6 Series	2017	Semi-Auto	18895	Diesel	145	51.5140	3.0	25998
4	18492	bmw	6 Series	2015	Automatic	62953	Diesel	160	51.4903	3.0	18990

```
X = train[['mileage', 'mpg', 'year', 'engineSize']]
Xtest = test[['mileage', 'mpg', 'year', 'engineSize']]
y = train['price']
ytest = test['price']
```

Let us make a bunch of small trees with bagging, so that we can visualize and see if they are being dominated by a particular predictor or predictor(s).

```
#Bagging the results of 10 decision trees to predict car price
model = BaggingRegressor(estimator=DecisionTreeRegressor(max_depth=3), n_estimators=10, random_state=42,
                        n_jobs=-1).fit(X, y)
```

```
#Change the index of model.estimators_[index] to visualize the 10 bagged trees, one at a time
dot_data = StringIO()
export_graphviz(model.estimators_[0], out_file=dot_data,
                filled=True, rounded=True,
                feature_names = ['mileage', 'mpg', 'year', 'engineSize'], precision=0)
graph = pydotplus.graph_from_dot_data(dot_data.getvalue())
#graph.write_png('car_price_tree.png')
Image(graph.create_png())
```



Each of the 10 bagged trees seems to be dominated by the `engineSize` predictor, thereby making the trees highly correlated. Average of highly correlated random variables has a higher variance than the average of lesser correlated random variables. Thus, highly correlated trees will tend to have a relatively high prediction variance despite averaging their predictions.

```
#Feature importance can be found by averaging the feature importance in individual trees
feature_importances = np.mean([
    tree.feature_importances_ for tree in model.estimators_
], axis=0)
feature_importances
```

```
array([0.13058631, 0.03965966, 0.22866077, 0.60109325])
```

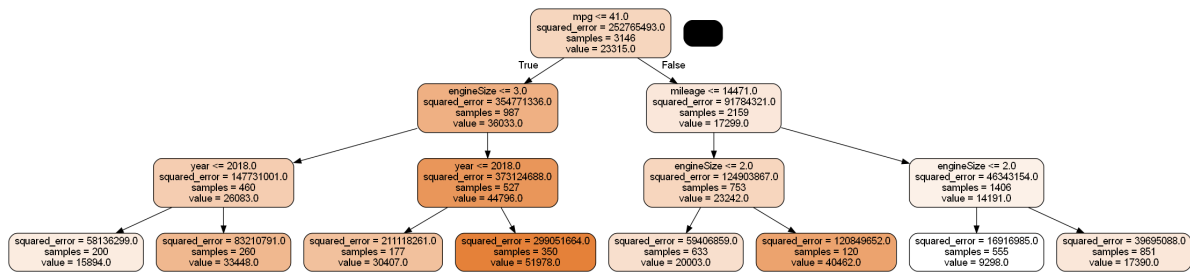
We can see that `engineSize` has the highest importance among predictors, supporting the visualization that it dominates the trees.

## 8.1 Random Forest for regression

Now, let us visualize small trees with the random forest algorithm to see if a predictor dominates all the trees.

```
#Averaging the results of 10 decision trees, while randomly considering sqrt(4)=2 predictors
#to split, to predict car price
model = RandomForestRegressor(n_estimators=10, random_state=1,max_features="sqrt",max_depth=5,
                             n_jobs=-1).fit(X, y)
```

```
#Change the index of model.estimators_[index] to visualize the 10 random forest trees, one at a time
dot_data = StringIO()
export_graphviz(model.estimators_[4], out_file=dot_data,
                filled=True, rounded=True,
                feature_names = ['mileage','mpg','year','engineSize'],precision=0)
graph = pydotplus.graph_from_dot_data(dot_data.getvalue())
#graph.write_png('car_price_tree.png')
Image(graph.create_png())
```



As two of the four predictors are randomly selected for splitting each node, **engineSize** no longer seems to dominate the trees. This will tend to reduce correlation among trees, thereby reducing the prediction variance, which in turn will tend to improve prediction accuracy.

```
#Averaging the results of 10 decision trees, while randomly considering sqrt(4)=2 predictors
#to split, to predict car price
model = RandomForestRegressor(n_estimators=10, random_state=1, max_features="sqrt",
                              n_jobs=-1).fit(X, y)
```

```
model.feature_importances_
```

```
array([0.16370584, 0.35425511, 0.18552673, 0.29651232])
```

Note that the feature importance of **engineSize** is reduced in random forests (as compared to bagged trees), and it no longer dominates the trees.

```
np.sqrt(mean_squared_error(test.price, model.predict(Xtest)))
```

```
5856.022395768459
```

The RMSE is similar to that obtained by bagging. We will discuss the comparison later.

### 8.1.1 Model accuracy vs number of trees

How does the model accuracy vary with the number of trees?

As we increase the number of trees, it will tend to reduce the variance of individual trees leading to a more accurate prediction.

```

#Finding model accuracy vs number of trees
warnings.filterwarnings("ignore")
oob_rsquared={};test_rsquared={};oob_rmse={};test_rmse = {}

for i in np.linspace(10,400,40, dtype=int):
    model = RandomForestRegressor(n_estimators=i, random_state=1,max_features="sqrt",
                                  n_jobs=-1,oob_score=True).fit(X, y)
    oob_rsquared[i]=model.oob_score_ #Returns the out-of_bag R-squared of the model
    test_rsquared[i]=model.score(Xtest,ytest) #Returns the test R-squared of the model
    oob_rmse[i]=np.sqrt(mean_squared_error(model.oob_prediction_,y))
    test_rmse[i]=np.sqrt(mean_squared_error(model.predict(Xtest),ytest))

warnings.resetwarnings()

# The hidden warning is: "Some inputs do not have OOB scores. This probably means too few
# estimators were used to compute any reliable oob estimates." This warning will appear
# in case of small number of estimators. In such a case, some observations may be use
# by all the estimators, and their OOB score can't be computed

```

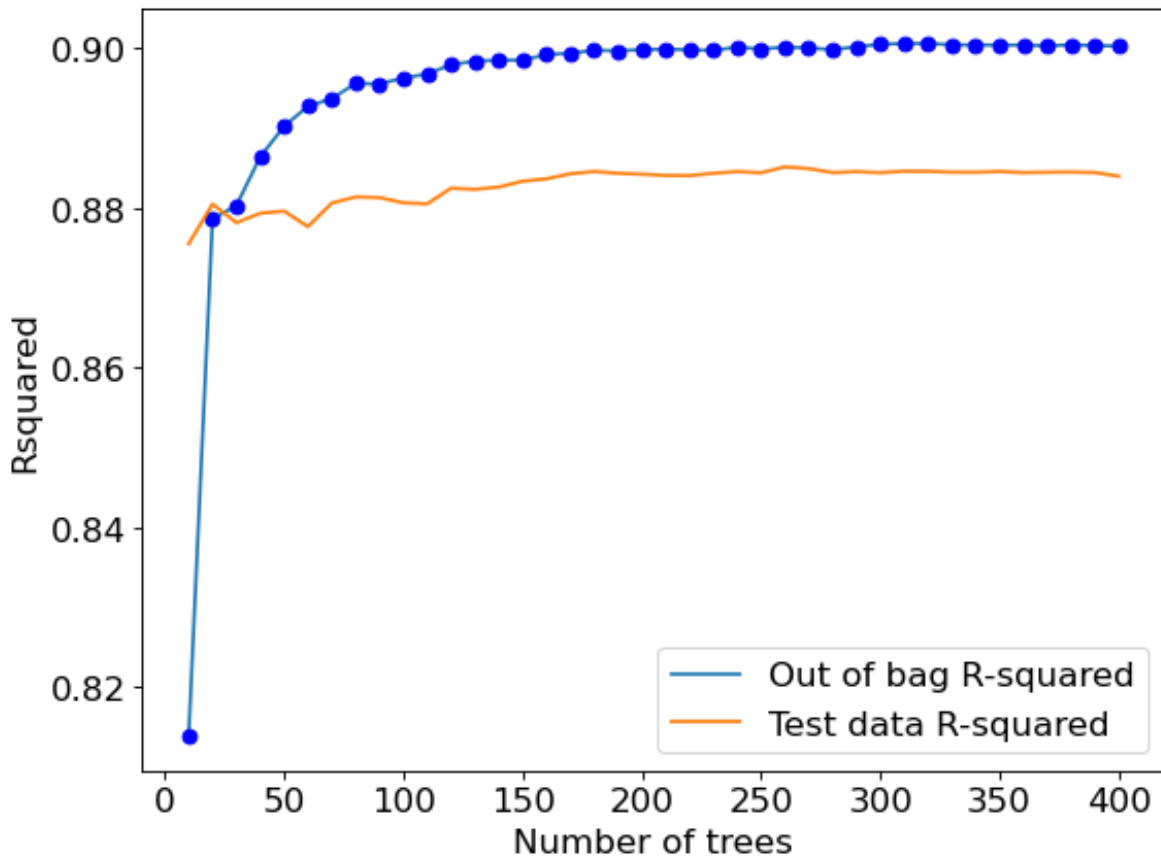
As we are ensemble only 10 trees in the first iteration, some of the observations are selected in every bootstrapped sample, and thus they don't have an out-of-bag error, which is producing the warning. For every observation to have an out-of-bag error, the number of trees must be sufficiently large.

Let us visualize the out-of-bag (OOB) R-squared and R-squared on test data vs the number of trees.

```

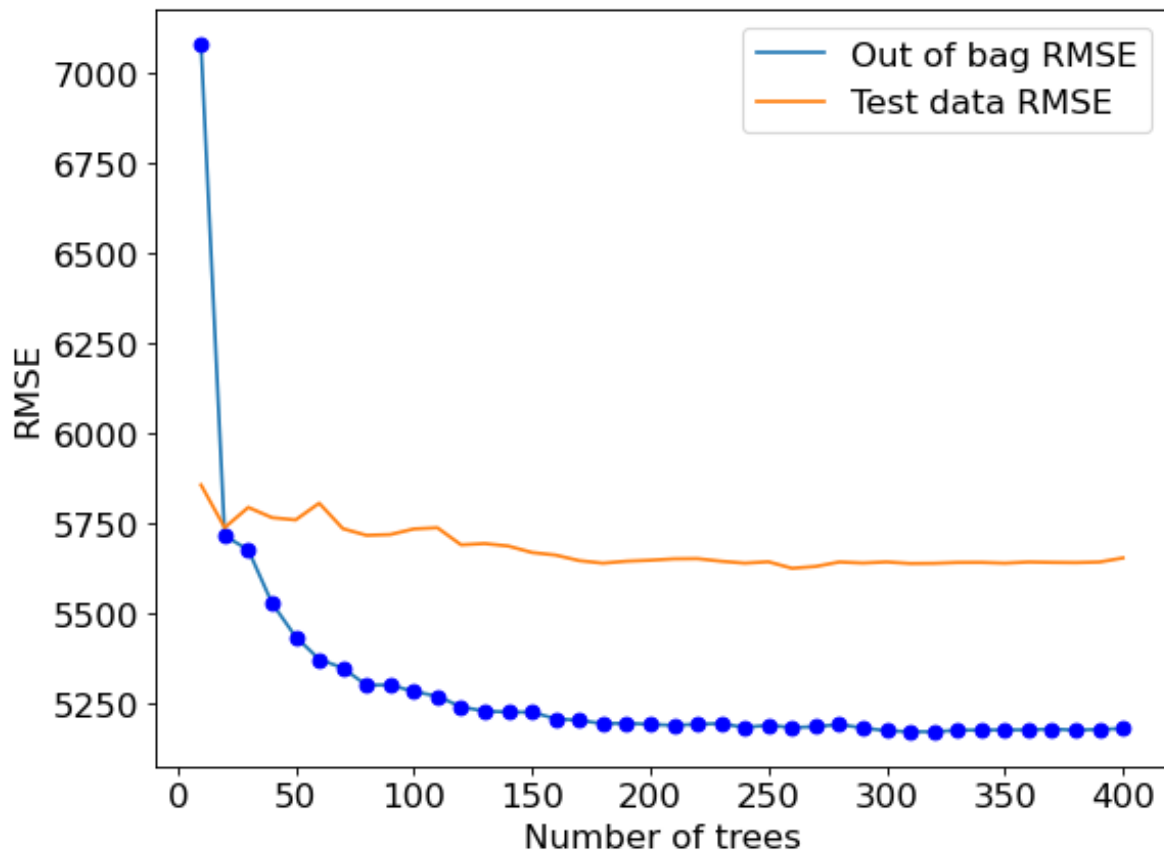
plt.rcParams.update({'font.size': 15})
plt.figure(figsize=(8, 6), dpi=80)
plt.plot(oob_rsquared.keys(),oob_rsquared.values(),label = 'Out of bag R-squared')
plt.plot(oob_rsquared.keys(),oob_rsquared.values(),'o',color = 'blue')
plt.plot(test_rsquared.keys(),test_rsquared.values(), label = 'Test data R-squared')
plt.xlabel('Number of trees')
plt.ylabel('Rsquared')
plt.legend();

```



The out-of-bag  $R$ -squared initially increases, and then stabilizes after a certain number of trees (around 200 in this case). Note that increasing the number of trees further will not lead to overfitting. However, increasing the number of trees will increase the computations. Thus, the number of trees developed should be the number beyond which the  $R$ -squared stabilizes.

```
#Visualizing out-of-bag RMSE and test data RMSE
plt.rcParams.update({'font.size': 15})
plt.figure(figsize=(8, 6), dpi=80)
plt.plot(oob_rmse.keys(),oob_rmse.values(),label = 'Out of bag RMSE')
plt.plot(oob_rmse.keys(),oob_rmse.values(),'o',color = 'blue')
plt.plot(test_rmse.keys(),test_rmse.values(), label = 'Test data RMSE')
plt.xlabel('Number of trees')
plt.ylabel('RMSE')
plt.legend();
```



A similar trend can be seen by plotting out-of-bag RMSE and test RMSE. Note that RMSE is proportional to R-squared. You only need to visualize one of RMSE or *R*-squared to find the optimal number of trees.

```
#Bagging with 150 trees
model = RandomForestRegressor(n_estimators=200, random_state=1,max_features="sqrt",
                             oob_score=True,n_jobs=-1).fit(X, y)
```

```
#OOB R-squared
model.oob_score_
```

```
0.8998265006519903
```

```
#RMSE on test data
pred = model.predict(Xtest)
np.sqrt(mean_squared_error(test.price, pred))
```

```
5647.195064555622
```



### 8.1.2 Tuning random forest

The Random forest object has options to set parameters such as depth, leaves, minimum number of observations in a leaf etc., for individual trees. These parameters are useful to prune a decision tree model consisting of a single tree, in order to avoid overfitting due to high variance of an unpruned tree.

Pruning individual trees in random forests is not likely to add much value, since averaging a sufficient number of unpruned trees reduces the variance of the trees, which enhances prediction accuracy. Pruning individual trees is unlikely to further reduce the prediction variance.

Here is a comment from page 596 of the [The Elements of Statistical Learning](#) that supports the above statement: *Segal (2004) demonstrates small gains in performance by controlling the depths of the individual trees grown in random forests. Our experience is that using full-grown trees seldom costs much, and results in one less tuning parameter.*

Below we attempt to optimize parameters that prune individual trees. However, as expected, it does not result in a substantial increase in prediction accuracy.

Also, note that we don't need to tune the number of trees in random forest with `GridSearchCV`. As we know the prediction accuracy will keep increasing with number of trees, we can tune the other hyperparameters with a constant value for the number of trees.

```
model.estimators_[0].get_n_leaves()
```

3086

```
model.estimators_[0].get_depth()
```

29

#### Coarse grid search

```
#Optimizing with OOB score takes half the time as compared to cross validation.
#The number of models developed with OOB score tuning is one-fifth of the number of models d
#5-fold cross validation
start_time = time.time()

n_samples = train.shape[0]
n_features = train.shape[1]

params = {'max_depth': [5, 10, 15, 20, 25, 30],
```

```

        'max_leaf_nodes':[600, 1200, 1800, 2400, 3000],
        'max_features': [1,2,3,4]}

param_list=list(it.product(*(params[Name] for Name in params)))

oob_score = [0]*len(param_list)
i=0
for pr in param_list:
    model = RandomForestRegressor(random_state=1,oob_score=True,verbose=False,
                                  n_estimators = 100, max_depth=pr[0],
                                  max_leaf_nodes=pr[1], max_features=pr[2], n_jobs=-1).fit(X,y)
    oob_score[i] = mean_squared_error(model.oob_prediction_, y, squared=False)
    i=i+1

end_time = time.time()
print("time taken = ", (end_time-start_time)/60, " minutes")
print("Best params = ", param_list[np.argmin(oob_score)])
print("Optimal OOB validation RMSE = ", np.min(oob_score))

```

```

time taken =  1.230358862876892  minutes
Best params =  (15, 1800, 3)
Optimal OOB validation RMSE =  5243.408784594606

```

### Finer grid search

Based on the coarse grid search, hyperparameters will be tuned in a finer grid around the optimal hyperparameter values obtained.

```

#Optimizing with OOB score takes half the time as compared to cross validation.
#The number of models developed with OOB score tuning is one-fifth of the number of models developed with cross validation.
#5-fold cross validation
start_time = time.time()

n_samples = train.shape[0]
n_features = train.shape[1]

params = {'max_depth': [12, 15, 18],
          'max_leaf_nodes':[1600, 1800, 2000],
          'max_features': [1,2,3,4]}

param_list=list(it.product(*(params[Name] for Name in params)))

```

```

oob_score = [0]*len(param_list)
i=0
for pr in param_list:
    model = RandomForestRegressor(random_state=1,oob_score=True,verbose=False,
                                  n_estimators = 100, max_depth=pr[0], max_leaf_nodes=pr[1],
                                  max_features=pr[2], n_jobs=-1).fit(X,y)
    oob_score[i] = mean_squared_error(model.oob_prediction_, y, squared=False)
    i=i+1

end_time = time.time()
print("time taken = ", (end_time-start_time)/60, " minutes")
print("Best params = ", param_list[np.argmin(oob_score)])
print("Optimal OOB validation RMSE = ", np.min(oob_score))

```

```

time taken = 0.4222299337387085  minutes
Best params = (15, 1800, 3)
Best score = 5243.408784594606

```

```

#Model with optimal parameters
model = RandomForestRegressor(n_estimators = 100, random_state=1, max_leaf_nodes = 1800, max.
                             oob_score=True,n_jobs=-1, max_features=3).fit(X, y)

```

```

#RMSE on test data
np.sqrt(mean_squared_error(test.price, model.predict(Xtest)))

```

```

5671.410705964455

```

Optimizing depth and leaves of individual trees didn't improve the prediction accuracy of the model. Important parameters to optimize in random forests will be the number of trees (`n_estimators`), and number of predictors considered at each split (`max_features`). However, sometimes individual pruning of trees may be useful. This may happen when the increase in bias in individual trees (*when pruned*) is lesser than the decrease in variance of the tree. However, if the pairwise correlation coefficient  $\rho$  of the trees increases by a certain extent on pruning, pruning may again be not useful.

```

#Tuning only n_estimators and max_features produces similar results
start_time = time.time()
params = {'max_features': [1,2,3,4]}

param_list=list(it.product(*(params[Name] for Name in params)))

```

```

oob_score = [0]*len(param_list)
i=0
for pr in param_list:
    model = RandomForestRegressor(random_state=1,oob_score=True,verbose=False,
                                  n_estimators = 100, max_features=pr[0], n_jobs=-1).fit(X,y)
    oob_score[i] = mean_squared_error(model.oob_prediction_, y, squared=False)
    i=i+1

end_time = time.time()
print("time taken = ", (end_time-start_time)/60, " minutes")
print("Best params = ", param_list[np.argmin(oob_score)])
print("Optimal OOB validation RMSE = ", np.min(oob_score))

```

```

time taken = 0.02856200933456421  minutes
Best params = (3,)
Best score (R-squared) = 5252.291978670057

```

```

#Model with optimal parameters
model = RandomForestRegressor(n_estimators=100, random_state=1,
                              n_jobs=-1, max_features=3).fit(X, y)
np.sqrt(mean_squared_error(test.price, model.predict(Xtest)))

```

```
5656.561522632323
```

Considering hyperparameters involving pruning, we observe a marginal decrease in the out-of-bag RMSE. Thus, other hyperparameters (*such as `max_features` and `max_samples`*) must be prioritized for tuning over hyperparameters involving pruning.

## 8.2 Random forest for classification

Random forest model to predict if a person has diabetes.

```

train = pd.read_csv('./Datasets/diabetes_train.csv')
test = pd.read_csv('./Datasets/diabetes_test.csv')

```

```

X = train.drop(columns = 'Outcome')
Xtest = test.drop(columns = 'Outcome')
y = train['Outcome']
ytest = test['Outcome']

```

```
#Ensembling the results of 10 decision trees
model = RandomForestClassifier(n_estimators=200, random_state=1,max_features="sqrt",n_jobs=-1)
```

```
#Feature importance for Random forest
np.mean([tree.feature_importances_ for tree in model.estimators_],axis=0)
```

```
array([0.08380406, 0.25403736, 0.09000104, 0.07151063, 0.07733353,
       0.16976023, 0.12289303, 0.13066012])
```

```
# Performance metrics computation for the optimum decision threshold probability
desired_threshold = 0.23
```

```
y_pred_prob = model.predict_proba(Xtest)[:,-1]
```

```
# Classifying observations in the positive class (y = 1) if the predicted probability is greater
# than the desired decision threshold probability
```

```
y_pred = y_pred_prob > desired_threshold
y_pred = y_pred.astype(int)
```

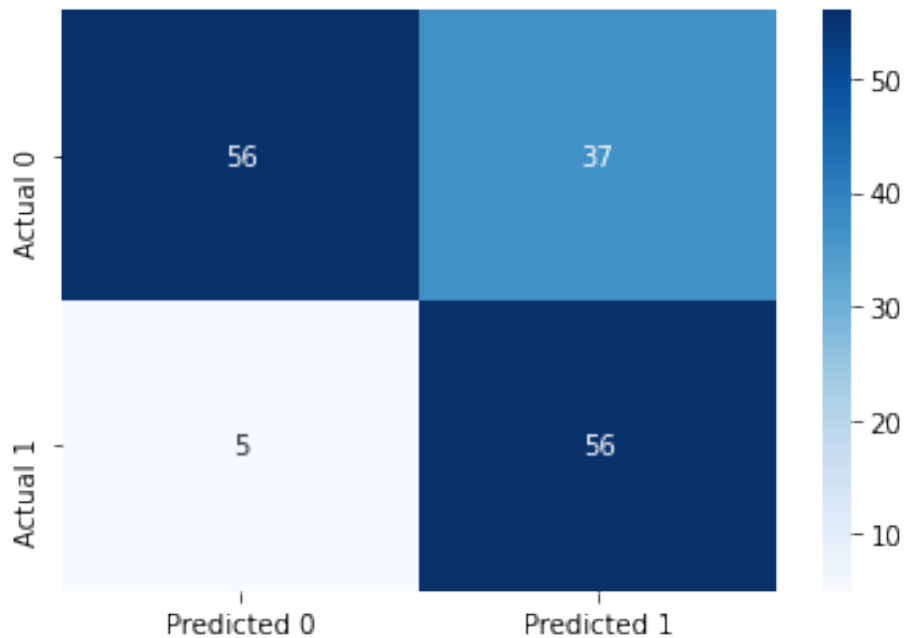
```
#Computing the accuracy
print("Accuracy: ",accuracy_score(y_pred, ytest)*100)
```

```
#Computing the ROC-AUC
fpr, tpr, auc_thresholds = roc_curve(ytest, y_pred_prob)
print("ROC-AUC: ",auc(fpr, tpr))# AUC of ROC
```

```
#Computing the precision and recall
print("Precision: ", precision_score(ytest, y_pred))
print("Recall: ", recall_score(ytest, y_pred))
```

```
#Confusion matrix
cm = pd.DataFrame(confusion_matrix(ytest, y_pred),
                  columns=['Predicted 0', 'Predicted 1'], index = ['Actual 0', 'Actual 1'])
sns.heatmap(cm, annot=True, cmap='Blues', fmt='g');
```

```
Accuracy: 72.72727272727273
ROC-AUC: 0.8744050766790058
Precision: 0.6021505376344086
Recall: 0.9180327868852459
```



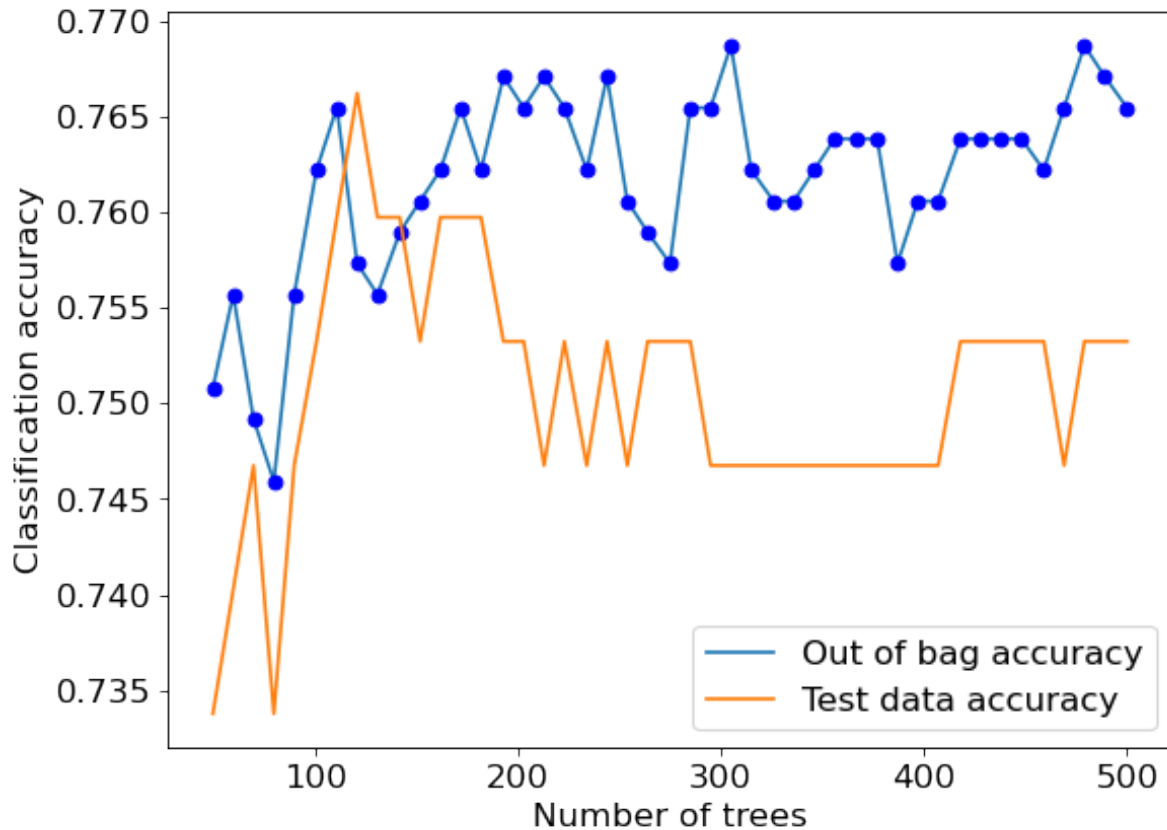
The model obtained above is similar to the one obtained by bagging. We'll discuss the comparison later.

### 8.2.1 Model accuracy vs number of trees

```
#Finding model accuracy vs number of trees
oob_accuracy={};test_accuracy={};oob_precision={}; test_precision = {}
for i in np.linspace(50,500,45,dtype=int):
    model = RandomForestClassifier(n_estimators=i, random_state=1,max_features="sqrt",n_jobs=
    oob_accuracy[i]=model.oob_score_ #Returns the out-of_bag R-squared of the model
    test_accuracy[i]=model.score(Xtest,ytest) #Returns the test R-squared of the model
    oob_pred = (model.oob_decision_function_[:,1]>=0.5).astype(int)
    oob_precision[i] = precision_score(y, oob_pred)
    test_pred = model.predict(Xtest)
    test_precision[i] = precision_score(ytest, test_pred)
```

```
plt.rcParams.update({'font.size': 15})
plt.figure(figsize=(8, 6), dpi=80)
plt.plot(oob_accuracy.keys(),oob_accuracy.values(),label = 'Out of bag accuracy')
plt.plot(oob_accuracy.keys(),oob_accuracy.values(),'o',color = 'blue')
plt.plot(test_accuracy.keys(),test_accuracy.values(), label = 'Test data accuracy')
```

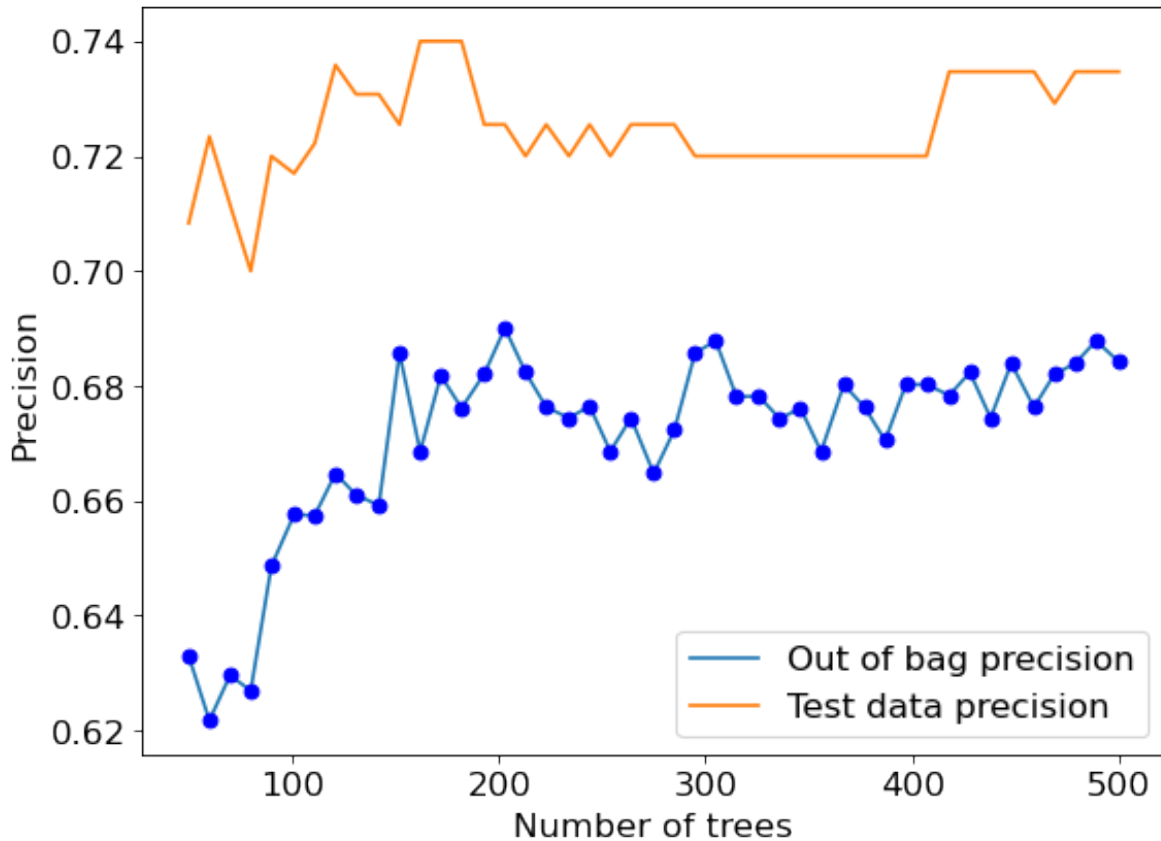
```
plt.xlabel('Number of trees')
plt.ylabel('Classification accuracy')
plt.legend();
```



We can also plot other metrics of interest such as out-of-bag precision vs number of trees.

```
#Precision vs number of trees
plt.rcParams.update({'font.size': 15})
plt.figure(figsize=(8, 6), dpi=80)
plt.plot(oob_precision.keys(),oob_precision.values(),label = 'Out of bag precision')
plt.plot(oob_precision.keys(),oob_precision.values(),'o',color = 'blue')
plt.plot(test_precision.keys(),test_precision.values(), label = 'Test data precision')

plt.xlabel('Number of trees')
plt.ylabel('Precision')
plt.legend();
```



## 8.2.2 Tuning random forest

Here we tune the number of predictors to be considered at each node for the split to maximize recall.

```
start_time = time.time()

params = {'n_estimators': [500],
          'max_features': range(1,9),
          }

param_list=list(it.product(*(params[Name] for Name in list(params.keys()))))
oob_recall = [0]*len(param_list)

i=0
for pr in param_list:
    model = RandomForestClassifier(random_state=1,oob_score=True,verbose=False,n_estimators =
```



```

max_features=pr[1], n_jobs=-1).fit(X,y)

oob_pred = (model.oob_decision_function_[:,1]>=0.5).astype(int)
oob_recall[i] = recall_score(y, oob_pred)
i=i+1

end_time = time.time()
print("time taken = ", (end_time-start_time)/60, " minutes")
print("max recall = ", np.max(oob_recall))
print("params= ", param_list[np.argmax(oob_recall)])

time taken = 0.08032723267873128 minutes
max recall = 0.5990338164251208
params= (500, 8)

model = RandomForestClassifier(random_state=1,n_jobs=-1,max_features=8,n_estimators=500).fit

# Performance metrics computation for the optimum decision threshold probability
desired_threshold = 0.23

y_pred_prob = model.predict_proba(Xtest)[:,-1]

# Classifying observations in the positive class (y = 1) if the predicted probability is greater
# than the desired decision threshold probability
y_pred = y_pred_prob > desired_threshold
y_pred = y_pred.astype(int)

#Computing the accuracy
print("Accuracy: ",accuracy_score(y_pred, ytest)*100)

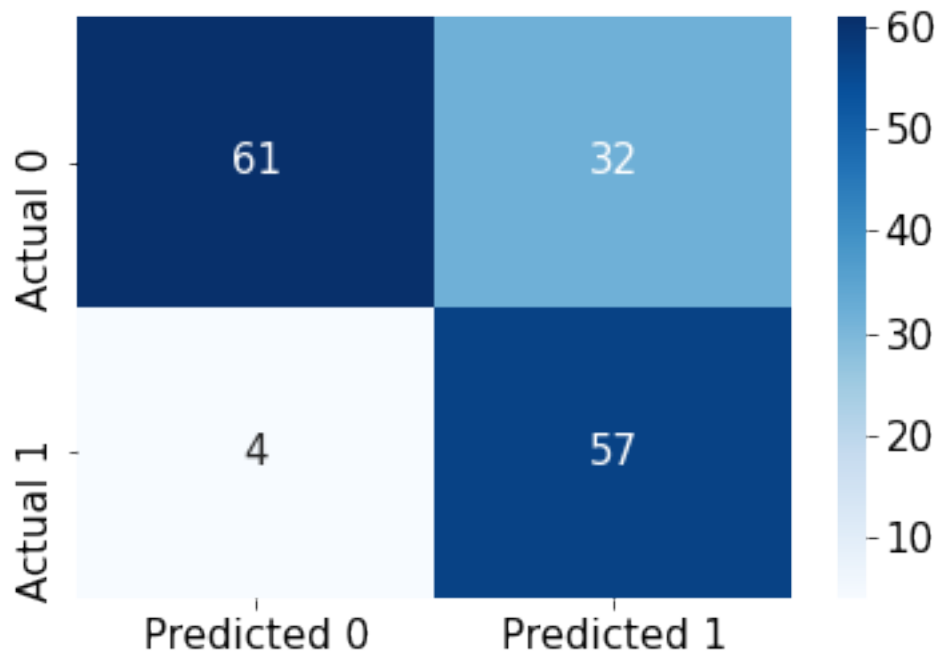
#Computing the ROC-AUC
fpr, tpr, auc_thresholds = roc_curve(ytest, y_pred_prob)
print("ROC-AUC: ",auc(fpr, tpr))# AUC of ROC

#Computing the precision and recall
print("Precision: ", precision_score(ytest, y_pred))
print("Recall: ", recall_score(ytest, y_pred))

#Confusion matrix
cm = pd.DataFrame(confusion_matrix(ytest, y_pred),
                  columns=['Predicted 0', 'Predicted 1'], index = ['Actual 0', 'Actual 1'])
sns.heatmap(cm, annot=True, cmap='Blues', fmt='g');

```

Accuracy: 76.62337662337663  
ROC-AUC: 0.8787237793054822  
Precision: 0.6404494382022472  
Recall: 0.9344262295081968



```
model.feature_importances_
```

```
array([0.069273, 0.31211579, 0.08492953, 0.05225877, 0.06179047,  
       0.17732674, 0.12342981, 0.1188759])
```

### 8.3 Random forest vs Bagging

We saw in the above examples that the performance of random forest was similar to that of bagged trees. This may happen in some cases including but not limited to:

1. All the predictors are more or less equally important, and the bagged trees are not highly correlated.

2. One of the predictors dominates the trees, resulting in highly correlated trees. However, each of the highly correlated trees have high prediction accuracy, leading to overall high prediction accuracy of the bagged trees despite the high correlation.

When can random forests perform poorly: When the number of variables is large, but the fraction of relevant variables small, random forests are likely to perform poorly with small  $m$  (fraction of predictors considered for each split). At each split the chance can be small that the relevant variables will be selected. - *Elements of Statistical Learning, page 596*.

However, in general, random forests are expected to decorrelate and improve the bagged trees.

Let us consider a classification example.

```
data = pd.read_csv('Heart.csv')
data.dropna(inplace = True)
data.head()
```

	Age	Sex	ChestPain	RestBP	Chol	Fbs	RestECG	MaxHR	ExAng	Oldpeak	Slope	Ca
0	63	1	typical	145	233	1	2	150	0	2.3	3	0.0
1	67	1	asymptomatic	160	286	0	2	108	1	1.5	2	3.0
2	67	1	asymptomatic	120	229	0	2	129	1	2.6	2	2.0
3	37	1	nonanginal	130	250	0	0	187	0	3.5	3	0.0
4	41	0	nontypical	130	204	0	2	172	0	1.4	1	0.0

In the above dataset, we wish to predict if a person has acquired heart disease (AHD = 'Yes'), based on their symptoms.

```
#Response variable
y = pd.get_dummies(data['AHD'])['Yes']

#Creating a dataframe for predictors with dummy variables replacing the categorical variables
X = data.drop(columns = ['AHD','ChestPain','Thal'])
X = pd.concat([X,pd.get_dummies(data['ChestPain']),pd.get_dummies(data['Thal'])],axis=1)
X.head()
```

	Age	Sex	RestBP	Chol	Fbs	RestECG	MaxHR	ExAng	Oldpeak	Slope	Ca	asymptomatic
0	63	1	145	233	1	2	150	0	2.3	3	0.0	0
1	67	1	160	286	0	2	108	1	1.5	2	3.0	1
2	67	1	120	229	0	2	129	1	2.6	2	2.0	1
3	37	1	130	250	0	0	187	0	3.5	3	0.0	0

	Age	Sex	RestBP	Chol	Fbs	RestECG	MaxHR	ExAng	Oldpeak	Slope	Ca	asymptomatic
4	41	0	130	204	0	2	172	0	1.4	1	0.0	0

```
X.shape
```

```
(297, 18)
```

```
#Creating train and test datasets
Xtrain,Xtest,ytrain,ytest = train_test_split(X,y,train_size = 0.5,random_state=1)
```

## Tuning random forest

```
#Tuning the random forest parameters
start_time = time.time()

oob_score = {}

i=0
for pr in range(1,19):
    model = RandomForestClassifier(random_state=1,oob_score=True,verbose=False,n_estimators = 100,
                                   max_features=pr, n_jobs=-1).fit(X,y)
    oob_score[i] = model.oob_score_
    i=i+1

end_time = time.time()
print("time taken = ", (end_time-start_time)/60, " minutes")
print("max accuracy = ", np.max(list(oob_score.values())))
print("Best value of max_features= ", np.argmax(list(oob_score.values()))+1)
```

```
time taken = 0.21557459433873494 minutes
max accuracy = 0.8249158249158249
Best value of max_features= 3
```

```
sns.scatterplot(x = oob_score.keys(),y = oob_score.values())
plt.xlabel('Max features')
plt.ylabel('Classification accuracy')
```

```
Text(0, 0.5, 'Classification accuracy')
```



Note that as the value of `max_features` is increasing, the accuracy is decreasing. This is probably due to the trees getting correlated as we consider more predictors for each split.

```
#Finding model accuracy vs number of trees
oob_accuracy={};test_accuracy={};
oob_accuracy2={};test_accuracy2={};

for i in np.linspace(100,500,40, dtype=int):
    #Bagging
    model = BaggingClassifier(base_estimator=DecisionTreeClassifier(), n_estimators=i, random_state=1,
                             n_jobs=-1,oob_score=True).fit(Xtrain, ytrain)
    oob_accuracy[i]=model.oob_score_ #Returns the out-of-bag classification accuracy of the model
    test_accuracy[i]=model.score(Xtest,ytest) #Returns the classification accuracy of the model

    #Random forest
    model2 = RandomForestClassifier(n_estimators=i, random_state=1,max_features=3,
                                   n_jobs=-1,oob_score=True).fit(Xtrain, ytrain)
    oob_accuracy2[i]=model2.oob_score_ #Returns the out-of-bag classification accuracy of the model
    test_accuracy2[i]=model2.score(Xtest,ytest) #Returns the classification accuracy of the model
```

```
#Feature importance for bagging
np.mean([tree.feature_importances_ for tree in model.estimators_],axis=0)
```

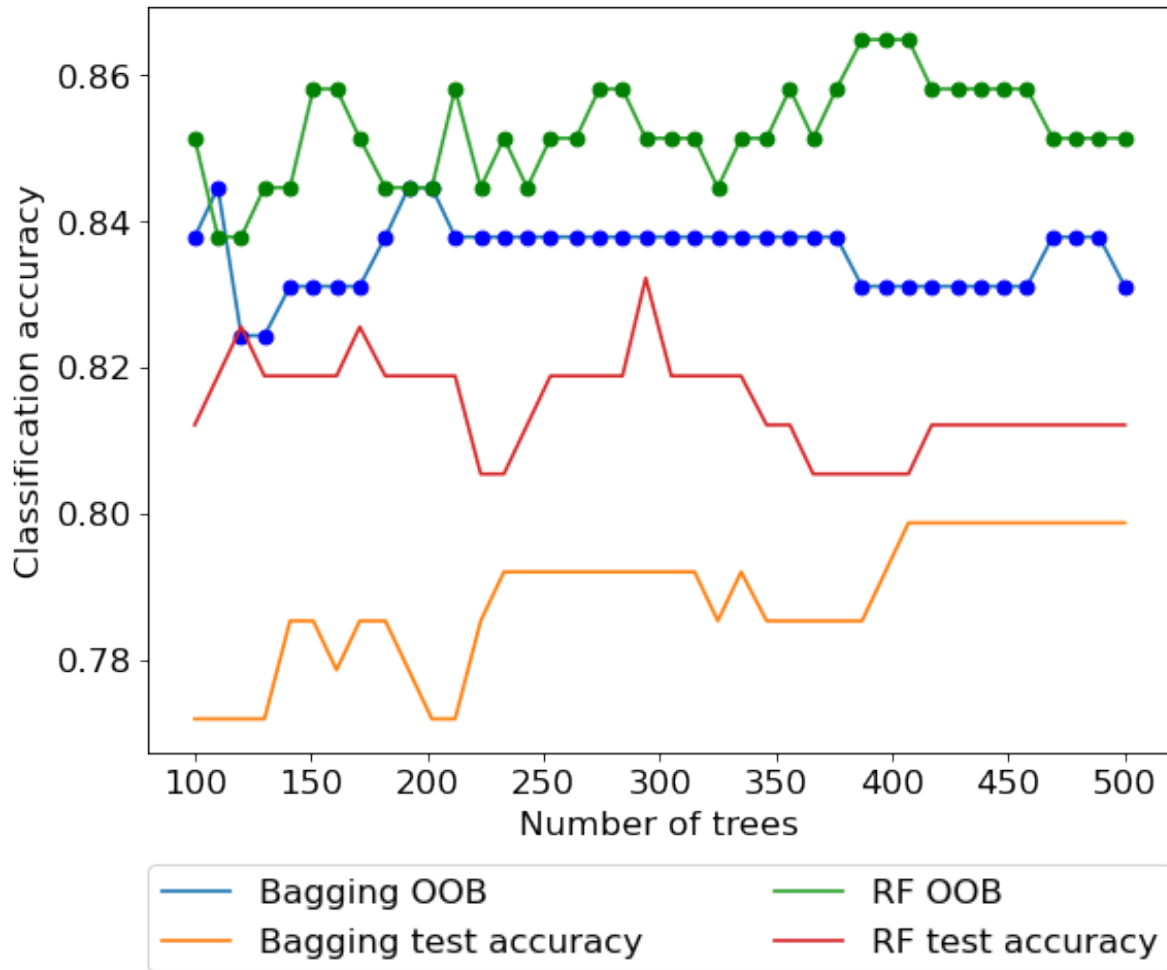
```
array([0.04381883, 0.05913479, 0.08585651, 0.07165678, 0.00302965,
       0.00903484, 0.05890448, 0.01223421, 0.072461  , 0.01337919,
       0.17495662, 0.18224651, 0.00527156, 0.00953965, 0.00396654,
       0.00163193, 0.09955286, 0.09332406])
```

Note that no predictor is too important to consider. That's why a small value of three for `max_features` is likely to decorrelate trees without compromising the quality of predictions.

```
plt.rcParams.update({'font.size': 15})
plt.figure(figsize=(8, 6), dpi=80)
plt.plot(oob_accuracy.keys(),oob_accuracy.values(),label = 'Bagging OOB')
plt.plot(oob_accuracy.keys(),oob_accuracy.values(),'o',color = 'blue')
plt.plot(test_accuracy.keys(),test_accuracy.values(), label = 'Bagging test accuracy')

plt.plot(oob_accuracy2.keys(),oob_accuracy2.values(),label = 'RF OOB')
plt.plot(oob_accuracy2.keys(),oob_accuracy2.values(),'o',color = 'green')
plt.plot(test_accuracy2.keys(),test_accuracy2.values(), label = 'RF test accuracy')

plt.xlabel('Number of trees')
plt.ylabel('Classification accuracy')
plt.legend(bbox_to_anchor=(0, -0.15, 1, 0), loc=2, ncol=2, mode="expand", borderaxespad=0)
```



In the above example we observe that random forest does improve over bagged trees in terms of classification accuracy. Unlike the previous two examples, the optimal value of `max_features` for random forests is much smaller than the total number of available predictors, thereby making the random forest model much different than the bagged tree model.

## 9 Adaptive Boosting

*Read section 8.2.3 of the book before using these notes.*

*Note that in this course, lecture notes are not sufficient, you must read the book for better understanding. Lecture notes are just implementing the concepts of the book on a dataset, but not explaining the concepts elaborately.*

For the exact algorithms underlying the AdaBoost algorithm, check out the papers [AdaBoostRegressor\(\)](#) and [AdaBoostClassifier\(\)](#).

### 9.1 Hyperparameters

There are 3 important parameters to tune in AdaBoost:

1. Number of trees
2. Depth of each tree
3. Learning rate

Let us visualize the accuracy of AdaBoost when we independently tweak each of the above parameters.

```
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import cross_val_score, train_test_split, KFold, cross_val_predict
from sklearn.metrics import mean_squared_error, r2_score, roc_curve, auc, precision_recall_curve
recall_score, precision_score, confusion_matrix
from sklearn.tree import DecisionTreeRegressor, DecisionTreeClassifier
from sklearn.model_selection import GridSearchCV, ParameterGrid, StratifiedKFold
from sklearn.ensemble import BaggingRegressor, BaggingClassifier, AdaBoostRegressor, AdaBoostClassifier
RandomForestRegressor
from sklearn.linear_model import LinearRegression, LogisticRegression
from sklearn.neighbors import KNeighborsRegressor
```



```
import itertools as it
import time as time

from skopt import BayesSearchCV
from skopt.space import Real, Categorical, Integer
from skopt.plots import plot_objective, plot_histogram, plot_convergence
import warnings
from IPython import display
```

```
#Using the same datasets as used for linear regression in STAT303-2,
#so that we can compare the non-linear models with linear regression
trainf = pd.read_csv('./Datasets/Car_features_train.csv')
trainp = pd.read_csv('./Datasets/Car_prices_train.csv')
testf = pd.read_csv('./Datasets/Car_features_test.csv')
testp = pd.read_csv('./Datasets/Car_prices_test.csv')
train = pd.merge(trainf, trainp)
test = pd.merge(testf, testp)
train.head()
```

	carID	brand	model	year	transmission	mileage	fuelType	tax	mpg	engineSize	price
0	18473	bmw	6 Series	2020	Semi-Auto	11	Diesel	145	53.3282	3.0	37980
1	15064	bmw	6 Series	2019	Semi-Auto	10813	Diesel	145	53.0430	3.0	33980
2	18268	bmw	6 Series	2020	Semi-Auto	6	Diesel	145	53.4379	3.0	36850
3	18480	bmw	6 Series	2017	Semi-Auto	18895	Diesel	145	51.5140	3.0	25998
4	18492	bmw	6 Series	2015	Automatic	62953	Diesel	160	51.4903	3.0	18990

```
X = train[['mileage', 'mpg', 'year', 'engineSize']]
Xtest = test[['mileage', 'mpg', 'year', 'engineSize']]
y = train['price']
ytest = test['price']
```

## 9.2 AdaBoost for regression

### 9.2.1 Number of trees vs cross validation error

As the number of trees increases, the prediction bias will decrease, and the prediction variance will increase. Thus, there will be an optimal number of trees that minimizes the prediction error.

```

def get_models():
    models = dict()
    # define number of trees to consider
    n_trees = [2, 5, 10, 50, 100, 500, 1000]
    for n in n_trees:
        models[str(n)] = AdaBoostRegressor(n_estimators=n, random_state=1)
    return models

# evaluate a given model using cross-validation
def evaluate_model(model, X, y):
    # define the evaluation procedure
    cv = KFold(n_splits=5, shuffle=True, random_state=1)
    # evaluate the model and collect the results
    scores = -cross_val_score(model, X, y, scoring='neg_root_mean_squared_error', cv=cv, n_jobs=5)
    return scores

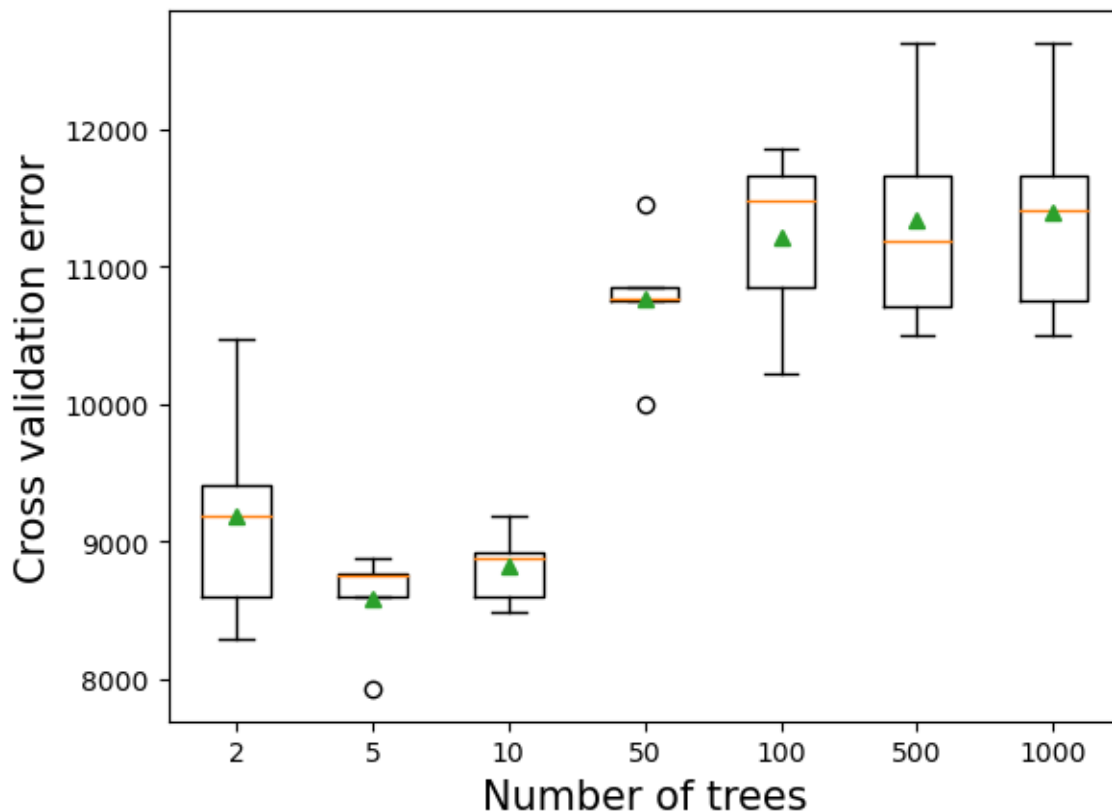
# get the models to evaluate
models = get_models()
# evaluate the models and store results
results, names = list(), list()
for name, model in models.items():
    # evaluate the model
    scores = evaluate_model(model, X, y)
    # store the results
    results.append(scores)
    names.append(name)
    # summarize the performance along the way
    print('>%s %.3f (%.3f)' % (name, np.mean(scores), np.std(scores)))
# plot model performance for comparison
plt.boxplot(results, labels=names, showmeans=True)
plt.ylabel('Cross validation error', fontsize=15)
plt.xlabel('Number of trees', fontsize=15);

```

```

>2 9190.253 (757.408)
>5 8583.629 (341.406)
>10 8814.328 (248.891)
>50 10763.138 (465.677)
>100 11217.783 (602.642)
>500 11336.088 (763.288)
>1000 11390.043 (752.446)

```



### 9.2.2 Depth of tree vs cross validation error

As the depth of each weak learner (decision tree) increases, the complexity of the weak learner will increase. As the complexity increases, the prediction bias will decrease, while the prediction variance will increase. Thus, there will be an optimal depth for each weak learner that minimizes the prediction error.

```
# get a list of models to evaluate
def get_models():
    models = dict()
    # explore depths from 1 to 10
    for i in range(1,21):
        # define base model
        base = DecisionTreeRegressor(max_depth=i)
        # define ensemble model
        models[str(i)] = AdaBoostRegressor(base_estimator=base,n_estimators=50)
    return models
```

```

# evaluate a given model using cross-validation
def evaluate_model(model, X, y):
    # define the evaluation procedure
    cv = KFold(n_splits=10, shuffle=True, random_state=1)
    # evaluate the model and collect the results
    scores = -cross_val_score(model, X, y, scoring='neg_root_mean_squared_error', cv=cv, n_jobs=1)
    return scores

# get the models to evaluate
models = get_models()
# evaluate the models and store results
results, names = list(), list()
for name, model in models.items():
    # evaluate the model
    scores = evaluate_model(model, X, y)
    # store the results
    results.append(scores)
    names.append(name)
    # summarize the performance along the way
    print('>%s %.3f (%.3f)' % (name, np.mean(scores), np.std(scores)))
plt.boxplot(results, labels=names, showmeans=True)
plt.ylabel('Cross validation error',fontsize=15)
plt.xlabel('Depth of each tree',fontsize=15);

```

```

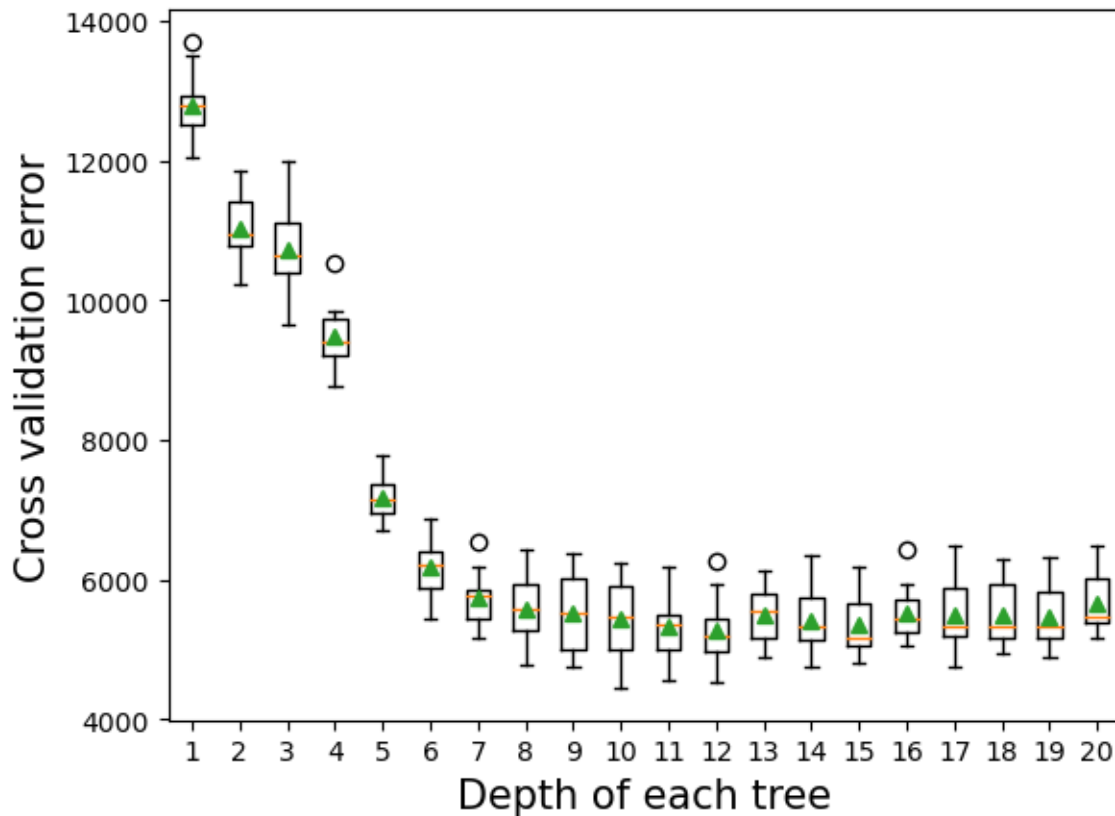
>1 12798.764 (490.538)
>2 11031.451 (465.520)
>3 10739.302 (636.517)
>4 9491.714 (466.764)
>5 7184.489 (324.484)
>6 6181.533 (411.394)
>7 5746.902 (407.451)
>8 5587.726 (473.619)
>9 5526.291 (541.512)
>10 5444.928 (554.170)
>11 5321.725 (455.899)
>12 5279.581 (492.785)
>13 5494.982 (393.469)
>14 5423.982 (488.564)
>15 5369.485 (441.799)
>16 5536.739 (409.166)
>17 5511.002 (517.384)

```

```

>18 5510.922 (478.285)
>19 5482.119 (465.565)
>20 5667.969 (468.964)

```



### 9.2.3 Learning rate vs cross validation error

The optimal learning rate will depend on the number of trees, and vice-versa. If the learning rate is too low, it will take several trees to “learn” the response. If the learning rate is high, the response will be “learned” quickly (with fewer) trees. Learning too quickly will be prone to overfitting, while learning too slowly will be computationally expensive. Thus, there will be an optimal learning rate to minimize the prediction error.

```

def get_models():
    models = dict()
    # explore learning rates from 0.1 to 2 in 0.1 increments
    for i in np.arange(0.1, 2.1, 0.1):
        key = '%.1f' % i

```

```

        models[key] = AdaBoostRegressor(learning_rate=i)
    return models

# evaluate a given model using cross-validation
def evaluate_model(model, X, y):
    # define the evaluation procedure
    cv = KFold(n_splits=10, shuffle=True, random_state=1)
    # evaluate the model and collect the results
    scores = -cross_val_score(model, X, y, scoring='neg_root_mean_squared_error', cv=cv, n_jobs=1)
    return scores

# get the models to evaluate
models = get_models()
# evaluate the models and store results
results, names = list(), list()
for name, model in models.items():
    # evaluate the model
    scores = evaluate_model(model, X, y)
    # store the results
    results.append(scores)
    names.append(name)
    # summarize the performance along the way
    print('>%s %.1f (%.1f)' % (name, np.mean(scores), np.std(scores)))
# plot model performance for comparison
plt.figure(figsize=(7, 7))
plt.boxplot(results, labels=names, showmeans=True)
plt.ylabel('Cross validation error',fontsize=15)
plt.xlabel('Learning rate',fontsize=15);

```

```

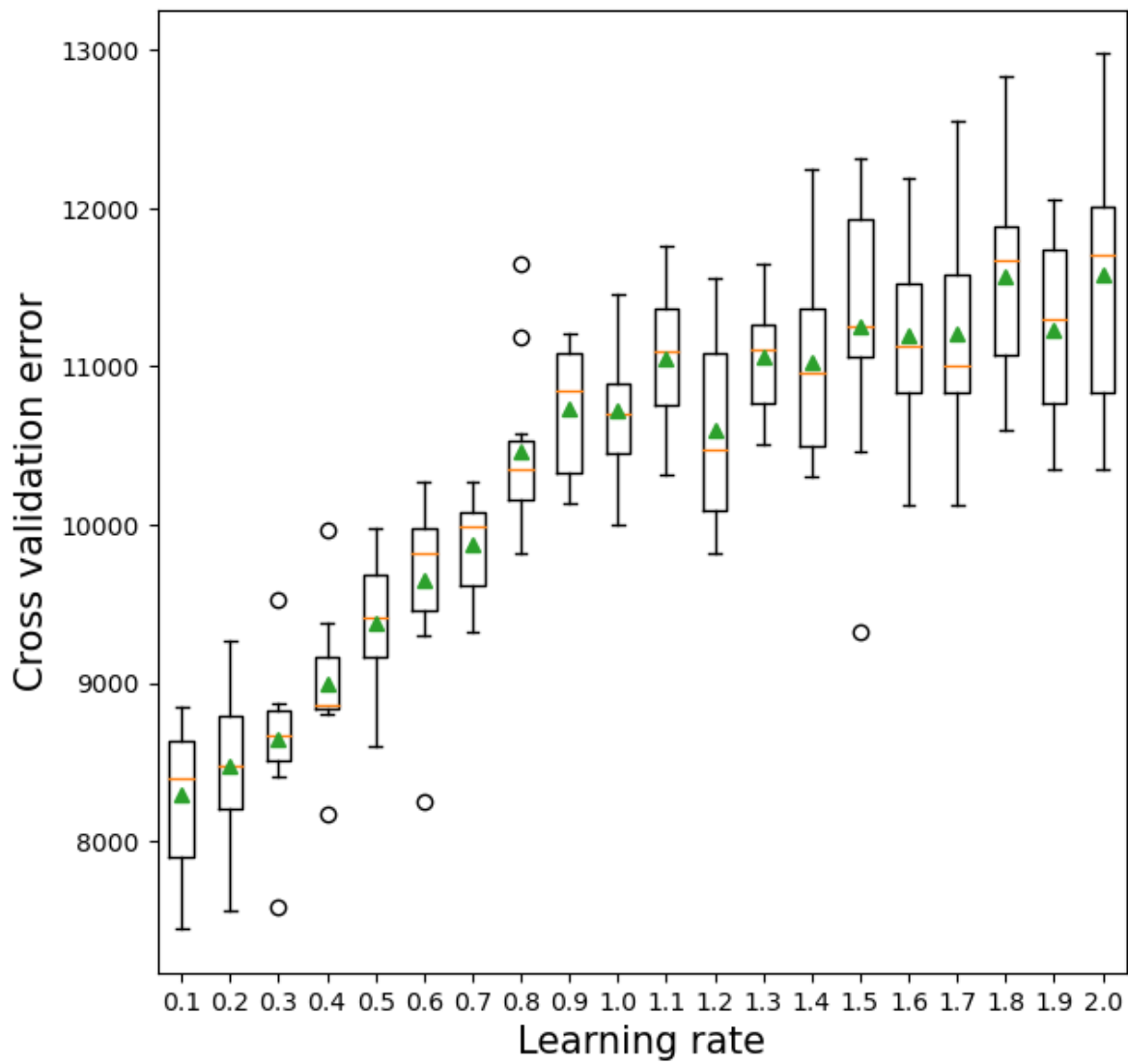
>0.1 8291.9 (452.4)
>0.2 8475.7 (465.3)
>0.3 8648.5 (458.8)
>0.4 8995.5 (438.6)
>0.5 9376.1 (388.2)
>0.6 9655.3 (551.8)
>0.7 9877.3 (319.8)
>0.8 10466.8 (528.3)
>0.9 10728.9 (386.8)
>1.0 10720.2 (410.6)
>1.1 11043.9 (432.5)
>1.2 10602.5 (570.0)
>1.3 11058.8 (362.1)

```

```

>1.4 11022.7 (616.0)
>1.5 11252.5 (839.3)
>1.6 11195.3 (604.5)
>1.7 11206.3 (636.1)
>1.8 11569.1 (674.6)
>1.9 11232.3 (605.6)
>2.0 11581.0 (824.8)

```



## 9.2.4 Tuning AdaBoost for regression

As the optimal value of the parameters depend on each other, we need to optimize them simultaneously.

```
model = AdaBoostRegressor(random_state=1)
grid = dict()
grid['n_estimators'] = [10, 50, 100, 200]
grid['learning_rate'] = [0.0001, 0.001, 0.01, 0.1, 1.0]
grid['estimator'] = [DecisionTreeRegressor(max_depth=3), DecisionTreeRegressor(max_depth=5),
                    DecisionTreeRegressor(max_depth=10), DecisionTreeRegressor(max_depth=15)]
# define the evaluation procedure
cv = KFold(n_splits=5, shuffle=True, random_state=1)
# define the grid search procedure
grid_search = GridSearchCV(estimator=model, param_grid=grid, n_jobs=-1, cv=cv, scoring='neg_mean_squared_error')
# execute the grid search
grid_result = grid_search.fit(X, y)
# summarize the best score and configuration
print("Best: %f using %s" % (-grid_result.best_score_, grid_result.best_params_))
# summarize all scores that were evaluated
means = grid_result.cv_results_['mean_test_score']
stds = grid_result.cv_results_['std_test_score']
params = grid_result.cv_results_['params']
```

Best: 5346.490675 using {'estimator': DecisionTreeRegressor(max\_depth=10), 'learning\_rate': 0.001}

Note that for tuning `max_depth` of the base estimator - decision tree, we specified 4 different base estimators with different depths. However, there is a more concise way to do that. We can specify the `max_depth` of the estimator by adding a double underscore “\_\_” between the `estimator` and the hyperparameter that we wish to tune (*max\_depth here*), and then specify its potential values in the `grid` itself as shown below. However, we’ll then need to add `DecisionTreeRegressor()` as the estimator within the `AdaBoostRegressor()` function.

```
model = AdaBoostRegressor(random_state=1, estimator = DecisionTreeRegressor(random_state=1))
grid = dict()
grid['n_estimators'] = [10, 50, 100, 200]
grid['learning_rate'] = [0.0001, 0.001, 0.01, 0.1, 1.0]
grid['estimator__max_depth'] = [3, 5, 10, 15]
# define the evaluation procedure
cv = KFold(n_splits=5, shuffle=True, random_state=1)
# define the grid search procedure
```



```

grid_search = GridSearchCV(estimator=model, param_grid=grid, n_jobs=-1, cv=cv, scoring='neg_
# execute the grid search
grid_result = grid_search.fit(X, y)
# summarize the best score and configuration
print("Best: %f using %s" % (-grid_result.best_score_, grid_result.best_params_))
# summarize all scores that were evaluated
means = grid_result.cv_results_['mean_test_score']
stds = grid_result.cv_results_['std_test_score']
params = grid_result.cv_results_['params']

```

Best: 5346.490675 using {'estimator\_\_max\_depth': 10, 'learning\_rate': 1.0, 'n\_estimators': 500}

The BayesSearchCV() approach also covers to a slightly different set of optimal hyperparameter values. However, it gives a similar cross-validated RMSE. This is possible. There may be multiple hyperparameter values that are different from each other, but similar in performance. It may be a good idea to ensemble models based on these two distinct set of hyperparameter values that give an equally accurate model.

```

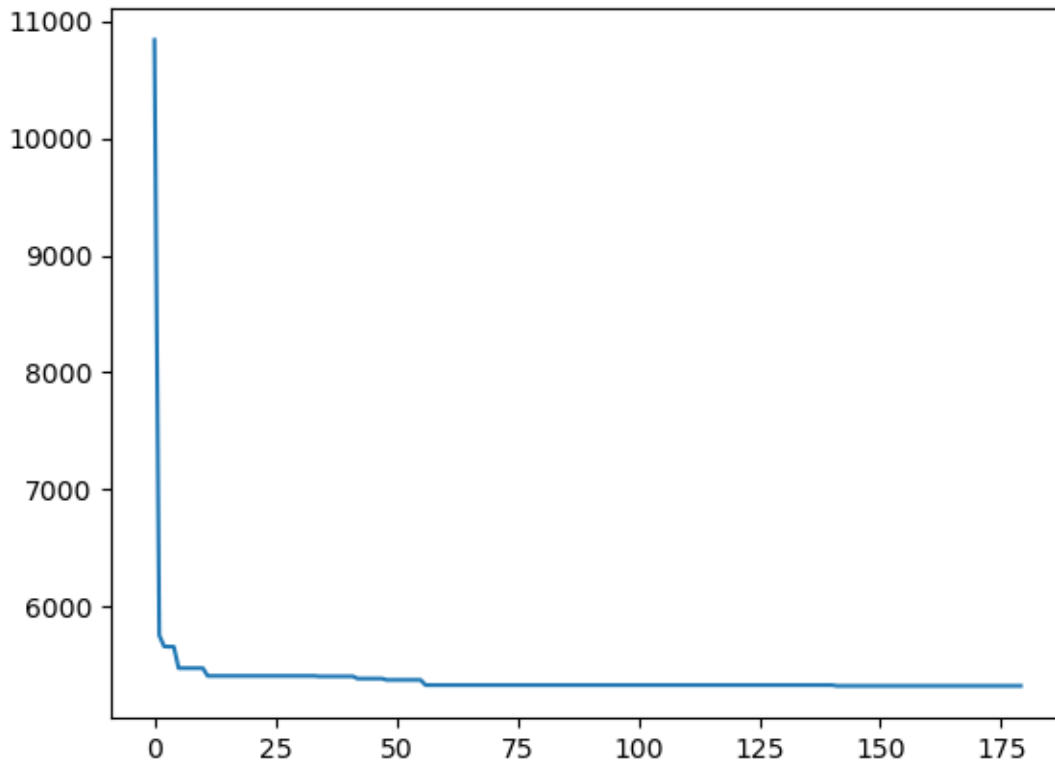
model = AdaBoostRegressor(estimator=DecisionTreeRegressor())
grid = dict()
grid['n_estimators'] = Integer(2, 1000)
grid['learning_rate'] = Real(0.0001, 1.0)
grid['estimator__max_depth'] = Integer(1, 20)

kfold = KFold(n_splits = 5, shuffle = True, random_state = 1)
gcv = BayesSearchCV(model, search_spaces = grid, cv = kfold, n_iter = 180, random_state = 10,
                    scoring = 'neg_root_mean_squared_error', n_jobs = -1)
paras = list(gcv.search_spaces.keys())
paras.sort()

def monitor(optim_result):
    cv_values = pd.Series(optim_result['func_vals']).cummin()
    display.clear_output(wait = True)
    min_ind = pd.Series(optim_result['func_vals']).argmin()
    print(paras, "=", optim_result['x_iters'][min_ind], pd.Series(optim_result['func_vals']))
    sns.lineplot(cv_values)
    plt.show()
gcv.fit(X, y, callback = monitor)

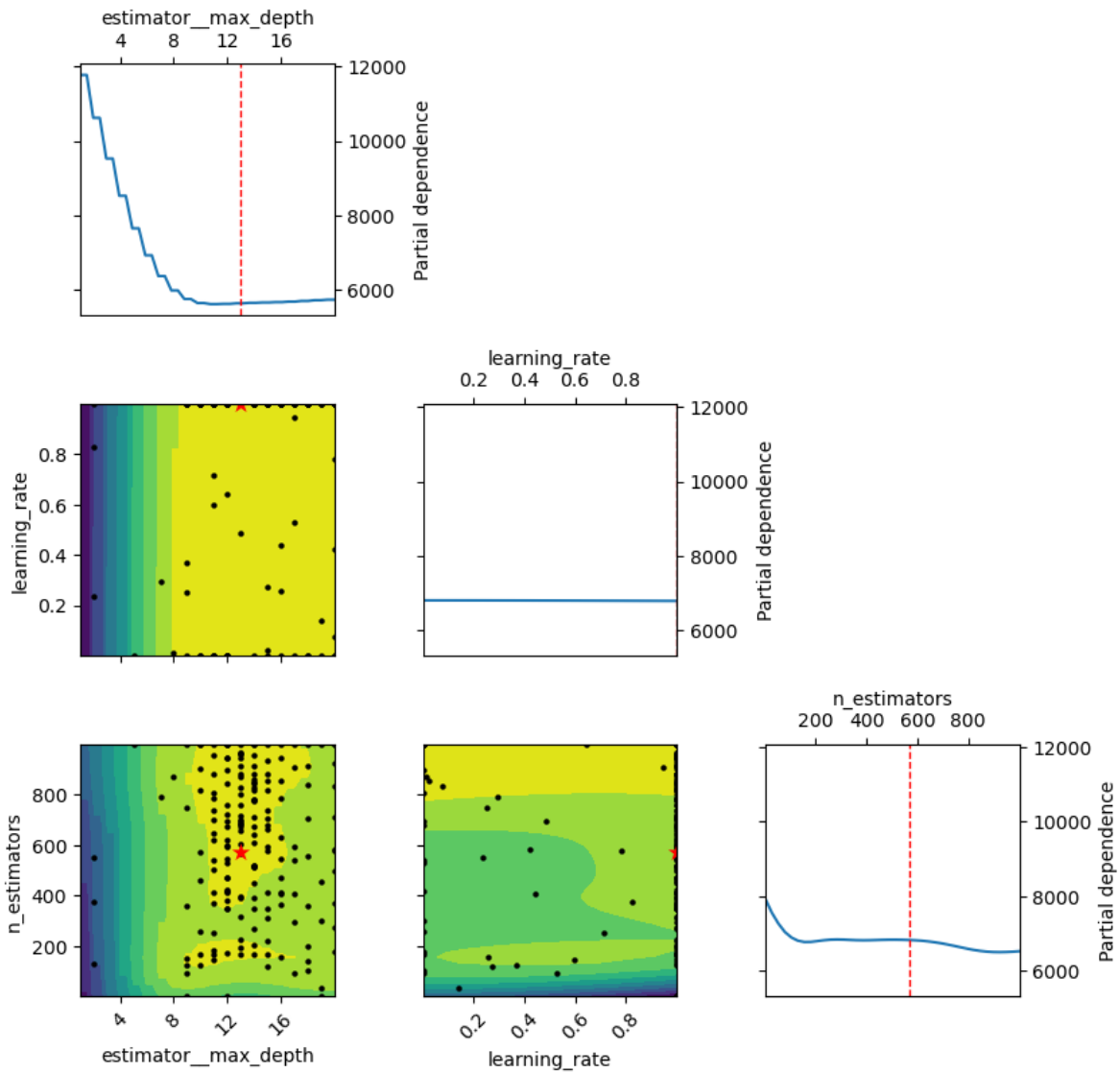
```

['estimator\_\_max\_depth', 'learning\_rate', 'n\_estimators'] = [13, 1.0, 570] 5325.017602505734

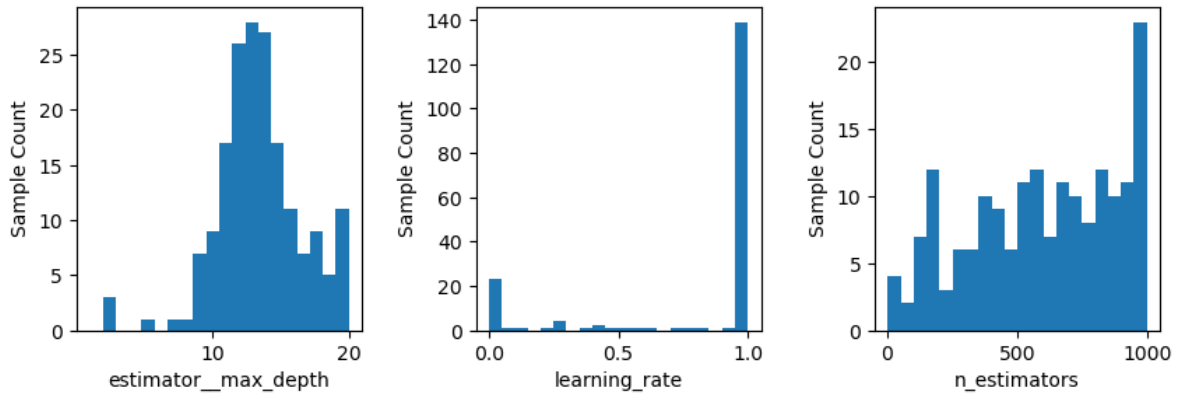


```
BayesSearchCV(cv=KFold(n_splits=5, random_state=1, shuffle=True),
               estimator=AdaBoostRegressor(estimator=DecisionTreeRegressor()),
               n_iter=180, n_jobs=-1, random_state=10,
               scoring='neg_root_mean_squared_error',
               search_spaces={'estimator__max_depth': Integer(low=1, high=20, prior='uniform',
                                                               'learning_rate': Real(low=0.0001, high=1.0, prior='uniform',
                                                               'n_estimators': Integer(low=2, high=1000, prior='uniform', trans

plot_objective(gcv.optimizer_results_[0],
               dimensions=['estimator__max_depth', 'learning_rate', 'n_estimators'], size=10,
               plt.show();
```



```
fig, ax = plt.subplots(1, 3, figsize = (10, 3))
plt.subplots_adjust(wspace=0.4)
plot_histogram(gcv.optimizer_results_[0], 0, ax = ax[0])
plot_histogram(gcv.optimizer_results_[0], 1, ax = ax[1])
plot_histogram(gcv.optimizer_results_[0], 2, ax = ax[2])
plt.show()
```



```
#Model based on the optimal hyperparameters
model = AdaBoostRegressor(estimator=DecisionTreeRegressor(max_depth=10),n_estimators=50,learning_rate=0.1,
                           random_state=1).fit(X,y)
```

```
#RMSE of the optimized model on test data
pred1=model.predict(Xtest)
print("AdaBoost model RMSE = ", np.sqrt(mean_squared_error(model.predict(Xtest),ytest)))
```

AdaBoost model RMSE = 5693.165811600585

```
#Model based on the optimal hyperparameters
model = AdaBoostRegressor(estimator=DecisionTreeRegressor(max_depth=13),n_estimators=570,learning_rate=0.1,
                           random_state=1).fit(X,y)
```

```
#RMSE of the optimized model on test data
pred2=model.predict(Xtest)
print("AdaBoost model RMSE = ", np.sqrt(mean_squared_error(model.predict(Xtest),ytest)))
```

AdaBoost model RMSE = 5434.852990644646

```
model = RandomForestRegressor(n_estimators=300, random_state=1,
                              n_jobs=-1, max_features=2).fit(X, y)
pred3 = model.predict(Xtest)
print("Random Forest model RMSE = ", np.sqrt(mean_squared_error(model.predict(Xtest),ytest)))
```

Random Forest model RMSE = 5642.45839697972

```
#Ensemble modeling
pred = 0.33*pred1+0.33*pred2 + 0.34*pred3
print("Ensemble model RMSE = ", np.sqrt(mean_squared_error(pred,ytest)))
```

Ensemble model RMSE = 5402.832128650372

Combined, the random forest model and the Adaboost models do better than each of the individual models.

## 9.3 AdaBoost for classification

Below is the AdaBoost implementation on a classification problem. The takeaways are the same as that of the regression problem above.

```
train = pd.read_csv('./Datasets/diabetes_train.csv')
test = pd.read_csv('./Datasets/diabetes_test.csv')
```

```
X = train.drop(columns = 'Outcome')
Xtest = test.drop(columns = 'Outcome')
y = train['Outcome']
ytest = test['Outcome']
```

### 9.3.1 Number of trees vs cross validation accuracy

```
def get_models():
    models = dict()
    # define number of trees to consider
    n_trees = [10, 50, 100, 500, 1000, 5000]
    for n in n_trees:
        models[str(n)] = AdaBoostClassifier(n_estimators=n,random_state=1)
    return models

# evaluate a given model using cross-validation
def evaluate_model(model, X, y):
    # define the evaluation procedure
    cv = StratifiedKFold(n_splits=10, shuffle=True, random_state=1)
    # evaluate the model and collect the results
```

```

    scores = cross_val_score(model, X, y, scoring='accuracy', cv=cv, n_jobs=-1)
    return scores

# get the models to evaluate
models = get_models()
# evaluate the models and store results
results, names = list(), list()
for name, model in models.items():
    # evaluate the model
    scores = evaluate_model(model, X, y)
    # store the results
    results.append(scores)
    names.append(name)
    # summarize the performance along the way
    print('>%s %.3f (%.3f)' % (name, np.mean(scores), np.std(scores)))
# plot model performance for comparison
plt.boxplot(results, labels=names, showmeans=True)
plt.ylabel('Cross validation error',fontsize=15)
plt.xlabel('Number of trees',fontsize=15)

```

```

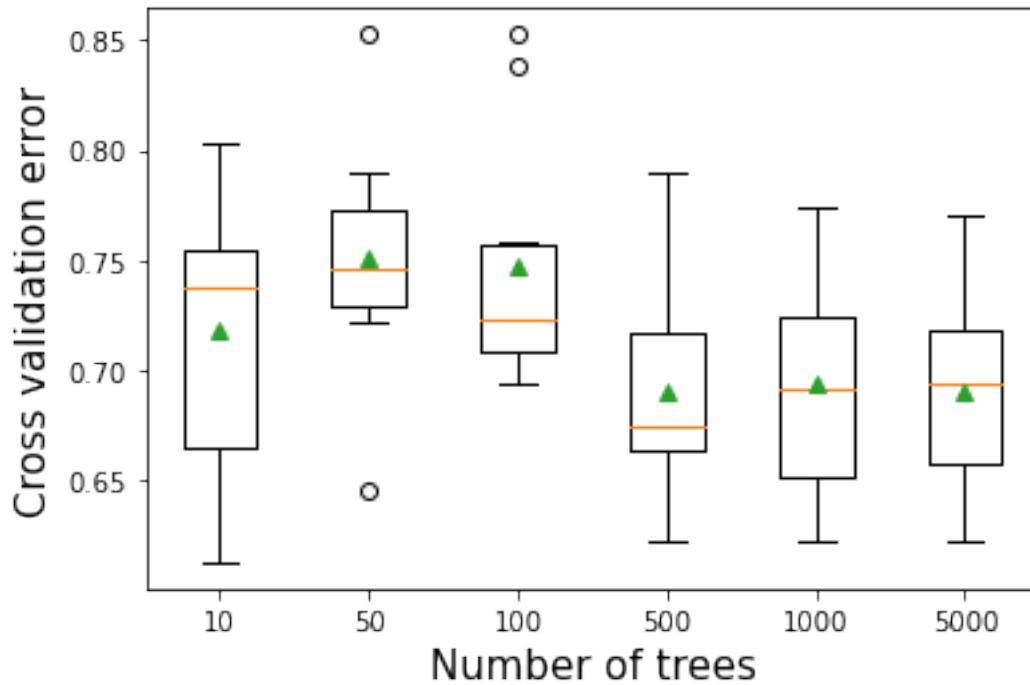
>10 0.718 (0.060)
>50 0.751 (0.051)
>100 0.748 (0.053)
>500 0.690 (0.045)
>1000 0.694 (0.048)
>5000 0.691 (0.044)

```

```

Text(0.5, 0, 'Number of trees')

```



### 9.3.2 Depth of each tree vs cross validation accuracy

```
# get a list of models to evaluate
def get_models():
    models = dict()
    # explore depths from 1 to 10
    for i in range(1,21):
        # define base model
        base = DecisionTreeClassifier(max_depth=i)
        # define ensemble model
        models[str(i)] = AdaBoostClassifier(estimator=base)
    return models

# evaluate a given model using cross-validation
def evaluate_model(model, X, y):
    # define the evaluation procedure
    cv = StratifiedKFold(n_splits=10, shuffle=True, random_state=1)
    # evaluate the model and collect the results
    scores = cross_val_score(model, X, y, scoring='accuracy', cv=cv, n_jobs=-1)
    return scores
```

```

# get the models to evaluate
models = get_models()
# evaluate the models and store results
results, names = list(), list()
for name, model in models.items():
    # evaluate the model
    scores = evaluate_model(model, X, y)
    # store the results
    results.append(scores)
    names.append(name)
    # summarize the performance along the way
    print('>%s %.3f (%.3f)' % (name, np.mean(scores), np.std(scores)))
plt.boxplot(results, labels=names, showmeans=True)
plt.ylabel('Accuracy',fontsize=15)
plt.xlabel('Depth of each tree',fontsize=15)

```

```

>1 0.751 (0.051)
>2 0.699 (0.063)
>3 0.696 (0.062)
>4 0.707 (0.055)
>5 0.713 (0.021)
>6 0.710 (0.061)
>7 0.733 (0.057)
>8 0.738 (0.044)
>9 0.727 (0.053)
>10 0.738 (0.065)
>11 0.748 (0.048)
>12 0.699 (0.044)
>13 0.738 (0.047)
>14 0.697 (0.041)
>15 0.697 (0.052)
>16 0.692 (0.052)
>17 0.702 (0.056)
>18 0.702 (0.045)
>19 0.700 (0.040)
>20 0.696 (0.042)

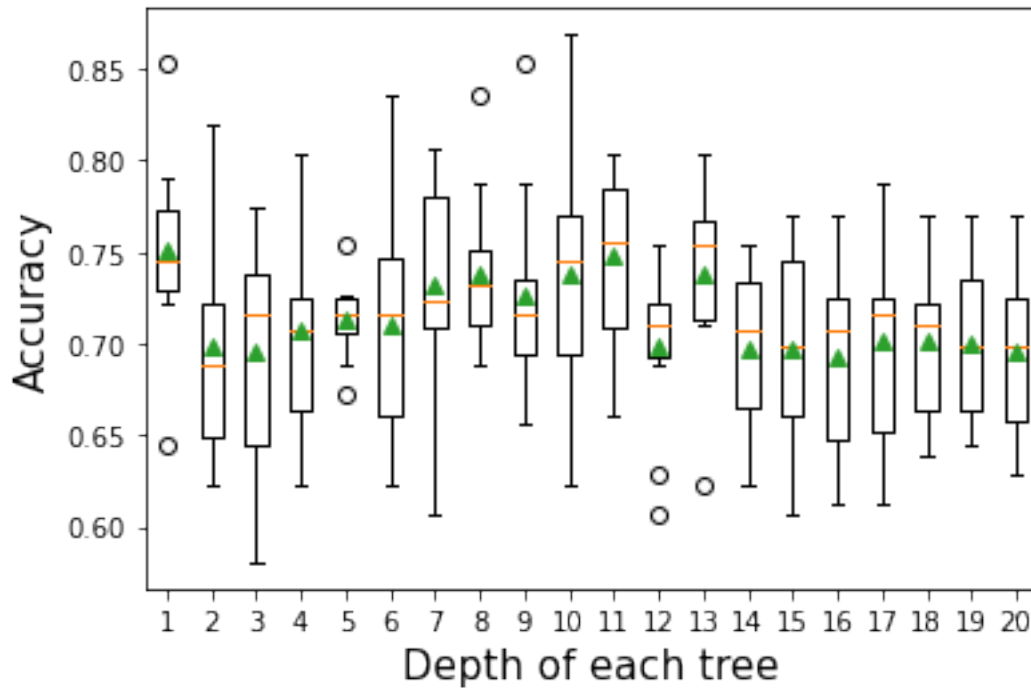
```

```

Text(0.5, 0, 'Depth of each tree')

```





### 9.3.3 Learning rate vs cross validation accuracy

```
def get_models():
    models = dict()
    # explore learning rates from 0.1 to 2 in 0.1 increments
    for i in np.arange(0.1, 2.1, 0.1):
        key = '%.1f' % i
        models[key] = AdaBoostClassifier(learning_rate=i)
    return models

# evaluate a given model using cross-validation
def evaluate_model(model, X, y):
    # define the evaluation procedure
    cv = KFold(n_splits=10, shuffle=True, random_state=1)
    # evaluate the model and collect the results
    scores = cross_val_score(model, X, y, scoring='accuracy', cv=cv, n_jobs=-1)
    return scores

# get the models to evaluate
models = get_models()
```

```

# evaluate the models and store results
results, names = list(), list()
for name, model in models.items():
    # evaluate the model
    scores = evaluate_model(model, X, y)
    # store the results
    results.append(scores)
    names.append(name)
    # summarize the performance along the way
    print('>%s %.3f (%.3f)' % (name, np.mean(scores), np.std(scores)))
# plot model performance for comparison
plt.figure(figsize=(7, 7))
plt.boxplot(results, labels=names, showmeans=True)
plt.ylabel('Accuracy',fontsize=15)
plt.xlabel('Learning rate',fontsize=15)

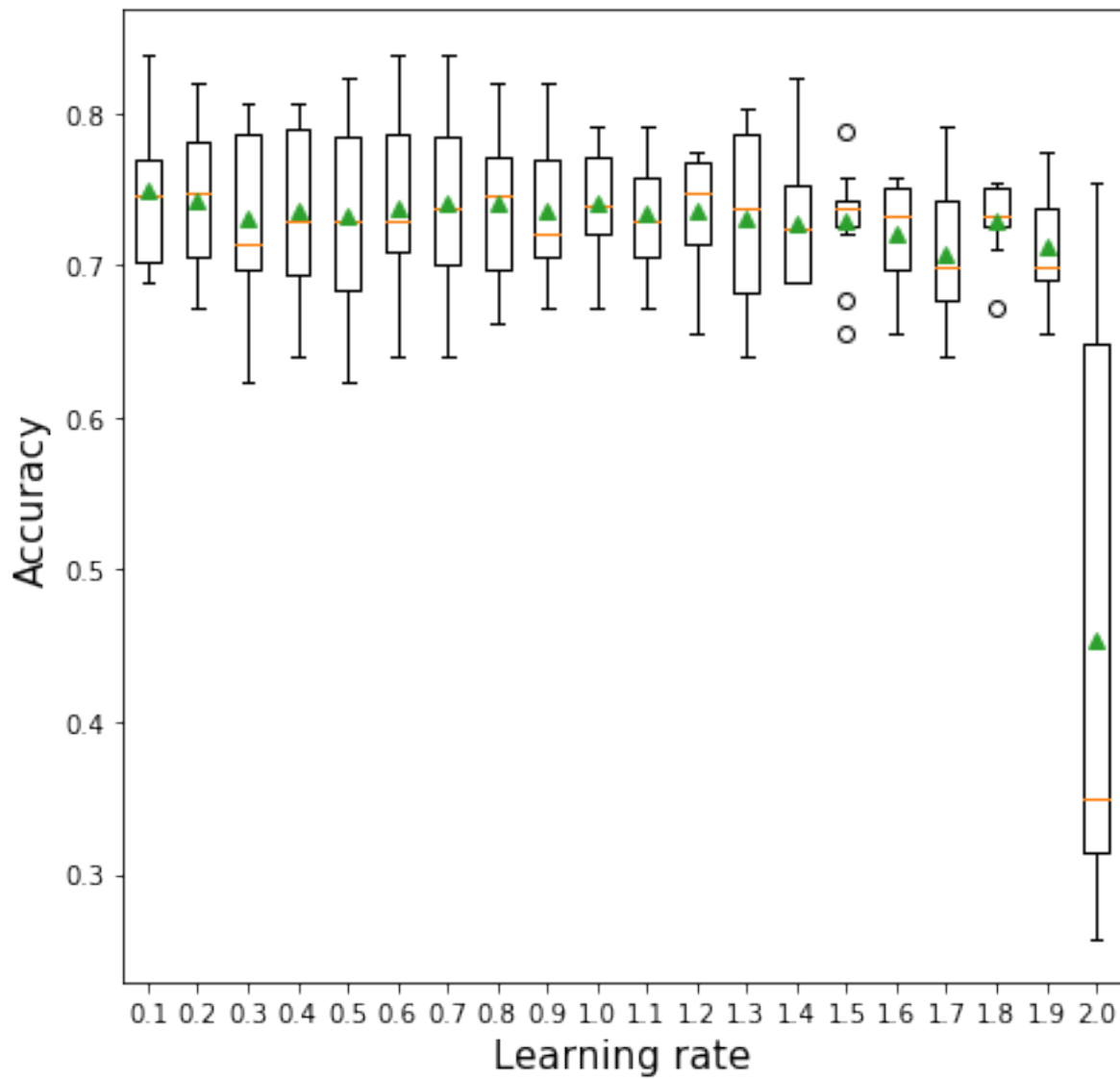
```

```

>0.1 0.749 (0.052)
>0.2 0.743 (0.050)
>0.3 0.731 (0.057)
>0.4 0.736 (0.053)
>0.5 0.733 (0.062)
>0.6 0.738 (0.058)
>0.7 0.741 (0.056)
>0.8 0.741 (0.049)
>0.9 0.736 (0.048)
>1.0 0.741 (0.035)
>1.1 0.734 (0.037)
>1.2 0.736 (0.038)
>1.3 0.731 (0.057)
>1.4 0.728 (0.041)
>1.5 0.730 (0.036)
>1.6 0.720 (0.038)
>1.7 0.707 (0.045)
>1.8 0.730 (0.024)
>1.9 0.712 (0.033)
>2.0 0.454 (0.191)

```

```
Text(0.5, 0, 'Learning rate')
```



### 9.3.4 Tuning AdaBoost Classifier hyperparameters

```
model = AdaBoostClassifier(random_state=1, estimator = DecisionTreeClassifier())
grid = dict()
grid['n_estimators'] = [10, 50, 100, 200, 500]
grid['learning_rate'] = [0.0001, 0.001, 0.01, 0.1, 1.0]
grid['estimator__max_depth'] = [1, 2, 3, 4]
# define the evaluation procedure
```

```

cv = StratifiedKFold(n_splits=5, shuffle=True, random_state=1)
# define the grid search procedure
grid_search = GridSearchCV(estimator=model, param_grid=grid, n_jobs=-1, cv=cv,
                           verbose = True)
# execute the grid search
grid_result = grid_search.fit(X, y)
# summarize the best score and configuration
print("Best: %f using %s" % (grid_result.best_score_, grid_result.best_params_))
# summarize all scores that were evaluated
means = grid_result.cv_results_['mean_test_score']
stds = grid_result.cv_results_['std_test_score']
params = grid_result.cv_results_['params']
#for mean, stdev, param in zip(means, stds, params):
#    print("%f (%f) with: %r" % (mean, stdev, param))

```

Fitting 5 folds for each of 100 candidates, totalling 500 fits

Best: 0.763934 using {'estimator\_\_max\_depth': 3, 'learning\_rate': 0.01, 'n\_estimators': 200}

### 9.3.5 Tuning the decision threshold probability

We'll find a decision threshold probability that balances recall with precision.

```

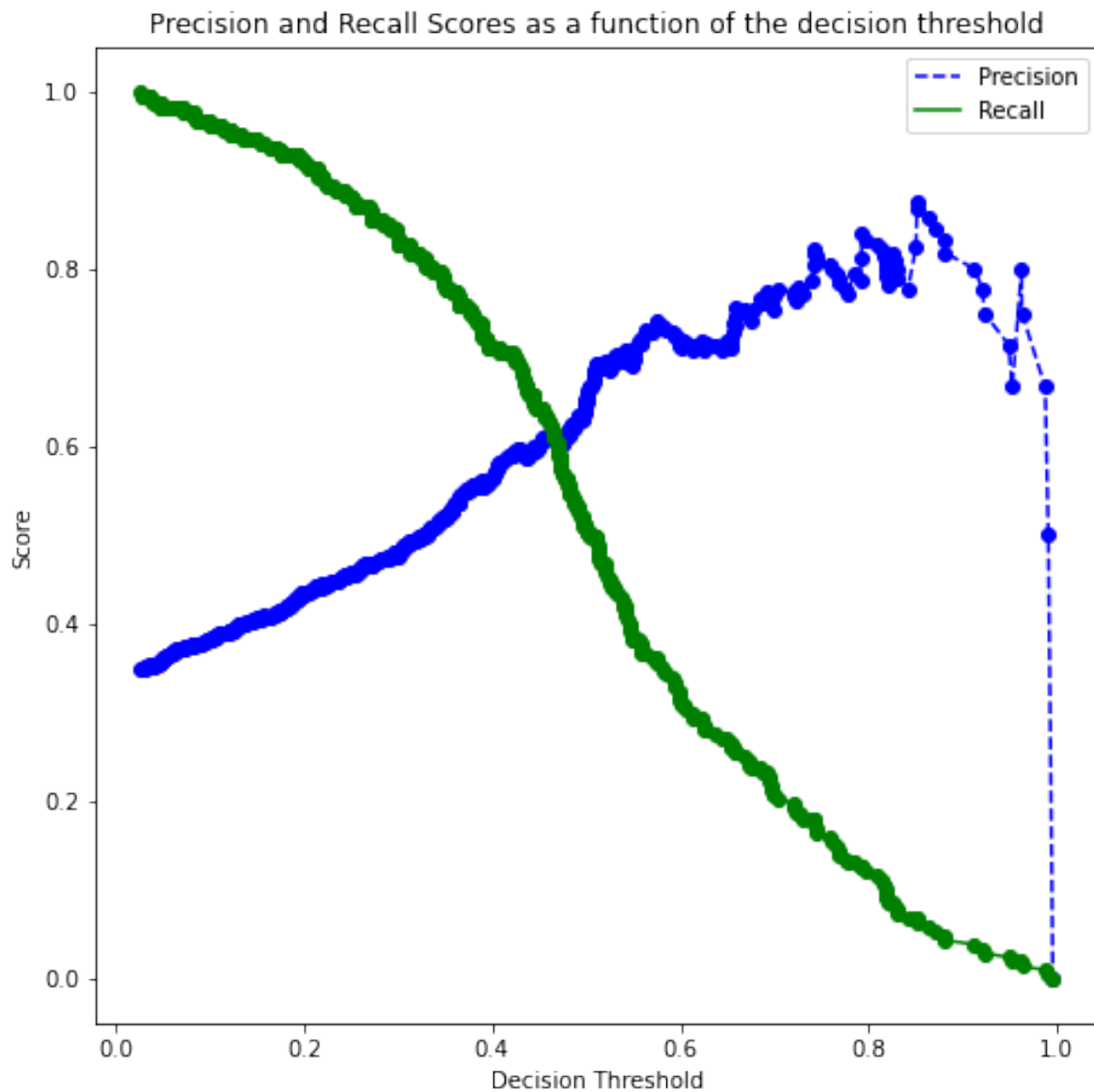
#Model based on the optimal parameters
model = AdaBoostClassifier(random_state=1, estimator = DecisionTreeClassifier(max_depth=3,
                                     n_estimators=200).fit(X,y)

# Note that we are using the cross-validated predicted probabilities, instead of directly using
# predicted probabilities on train data, as the model may be overfitting on the train data, and
# may lead to misleading results
cross_val_ypred = cross_val_predict(AdaBoostClassifier(random_state=1,base_estimator = DecisionTreeClassifier(
                                     n_estimators=200), X, y, cv = 5, method = 'predict_proba')

p, r, thresholds = precision_recall_curve(y, cross_val_ypred[:,1])
def plot_precision_recall_vs_threshold(precisions, recalls, thresholds):
    plt.figure(figsize=(8, 8))
    plt.title("Precision and Recall Scores as a function of the decision threshold")
    plt.plot(thresholds, precisions[:-1], "b--", label="Precision")
    plt.plot(thresholds, recalls[:-1], "g-", label="Recall")
    plt.plot(thresholds, precisions[:-1], "o", color = 'blue')
    plt.plot(thresholds, recalls[:-1], "o", color = 'green')
    plt.ylabel("Score")

```

```
plt.xlabel("Decision Threshold")
plt.legend(loc='best')
plt.legend()
plot_precision_recall_vs_threshold(p, r, thresholds)
```



```
# Thresholds with precision and recall
all_thresholds = np.concatenate([thresholds.reshape(-1,1), p[:-1].reshape(-1,1), r[:-1].reshape(-1,1)])
recall_more_than_80 = all_thresholds[all_thresholds[:,2]>0.8,:]
```

```
# As the values in 'recall_more_than_80' are arranged in decreasing order of recall and increasing order of precision,
# the last value will provide the maximum threshold probability for the recall to be more than 80%
# We wish to find the maximum threshold probability to obtain the maximum possible precision
recall_more_than_80[recall_more_than_80.shape[0]-1]
```

```
array([0.33488762, 0.50920245, 0.80193237])
```

```
#Optimal decision threshold probability
thres = recall_more_than_80[recall_more_than_80.shape[0]-1][0]
thres
```

```
0.3348876199649718
```

```
# Performance metrics computation for the optimum decision threshold probability
desired_threshold = thres

y_pred_prob = model.predict_proba(Xtest)[:,-1]

# Classifying observations in the positive class (y = 1) if the predicted probability is greater
# than the desired decision threshold probability
y_pred = y_pred_prob > desired_threshold
y_pred = y_pred.astype(int)

#Computing the accuracy
print("Accuracy: ",accuracy_score(y_pred, ytest)*100)

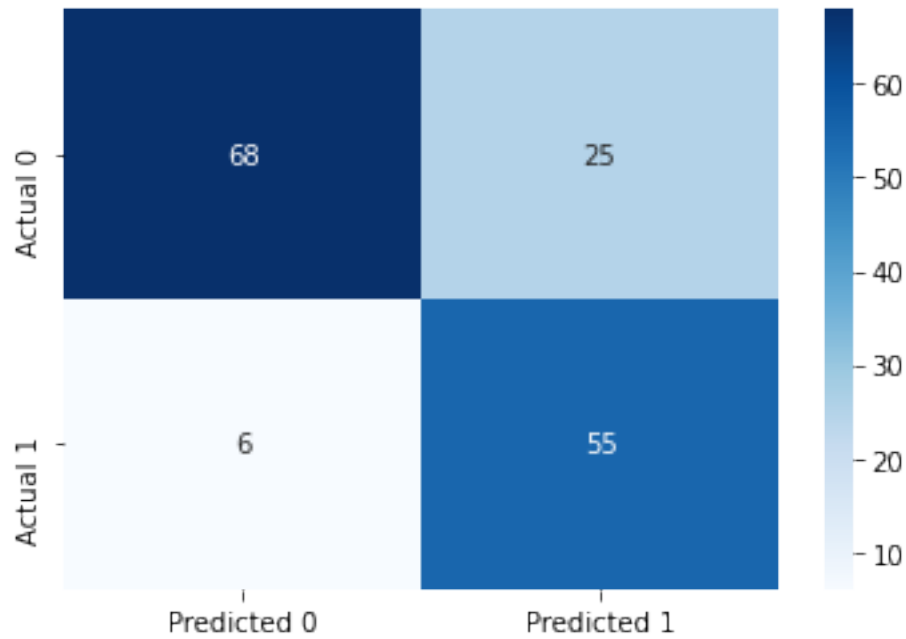
#Computing the ROC-AUC
fpr, tpr, auc_thresholds = roc_curve(ytest, y_pred_prob)
print("ROC-AUC: ",auc(fpr, tpr))# AUC of ROC

#Computing the precision and recall
print("Precision: ", precision_score(ytest, y_pred))
print("Recall: ", recall_score(ytest, y_pred))

#Confusion matrix
cm = pd.DataFrame(confusion_matrix(ytest, y_pred),
                  columns=['Predicted 0', 'Predicted 1'], index = ['Actual 0', 'Actual 1'])
sns.heatmap(cm, annot=True, cmap='Blues', fmt='g');
```

```
Accuracy: 79.87012987012987
```

RDC-AUC: 0.8884188260179798  
Precision: 0.6875  
Recall: 0.9016393442622951



The above model is similar to the one obtained with bagging / random forest. However, adaptive boosting may lead to better classification performance as compared to bagging / random forest.

# 10 Gradient Boosting

Check the gradient boosting algorithm in section 10.10.2 of the book, [Elements of Statistical Learning](#) before using these notes.

*Note that in this course, lecture notes are not sufficient, you must read the book for better understanding. Lecture notes are just implementing the concepts of the book on a dataset, but not explaining the concepts elaborately.*

## 10.1 Hyperparameters

There are 5 important parameters to tune in Gradient boosting:

1. Number of trees
2. Depth of each tree
3. Learning rate
4. Subsample fraction
5. Maximum features

Let us visualize the accuracy of Gradient boosting when we independently tweak each of the above parameters.

```
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import cross_val_score, train_test_split, KFold, cross_val_predict
from sklearn.metrics import mean_squared_error, r2_score, roc_curve, auc, precision_recall_curve
recall_score, precision_score, confusion_matrix
from sklearn.tree import DecisionTreeRegressor, DecisionTreeClassifier
from sklearn.model_selection import GridSearchCV, ParameterGrid, StratifiedKFold
from sklearn.ensemble import GradientBoostingRegressor, GradientBoostingClassifier, BaggingRegressor
from sklearn.linear_model import LinearRegression, LogisticRegression
```



```

from sklearn.neighbors import KNeighborsRegressor
import itertools as it
import time as time

from skopt import BayesSearchCV
from skopt.space import Real, Categorical, Integer
from skopt.plots import plot_objective, plot_histogram, plot_convergence
import warnings
from IPython import display

```

```

#Using the same datasets as used for linear regression in STAT303-2,
#so that we can compare the non-linear models with linear regression
trainf = pd.read_csv('./Datasets/Car_features_train.csv')
trainp = pd.read_csv('./Datasets/Car_prices_train.csv')
testf = pd.read_csv('./Datasets/Car_features_test.csv')
testp = pd.read_csv('./Datasets/Car_prices_test.csv')
train = pd.merge(trainf, trainp)
test = pd.merge(testf, testp)
train.head()

```

	carID	brand	model	year	transmission	mileage	fuelType	tax	mpg	engineSize	price
0	18473	bmw	6 Series	2020	Semi-Auto	11	Diesel	145	53.3282	3.0	37980
1	15064	bmw	6 Series	2019	Semi-Auto	10813	Diesel	145	53.0430	3.0	33980
2	18268	bmw	6 Series	2020	Semi-Auto	6	Diesel	145	53.4379	3.0	36850
3	18480	bmw	6 Series	2017	Semi-Auto	18895	Diesel	145	51.5140	3.0	25998
4	18492	bmw	6 Series	2015	Automatic	62953	Diesel	160	51.4903	3.0	18990

```

X = train[['mileage', 'mpg', 'year', 'engineSize']]
Xtest = test[['mileage', 'mpg', 'year', 'engineSize']]
y = train['price']
ytest = test['price']

```

## 10.2 Gradient boosting for regression

### 10.2.1 Number of trees vs cross validation error

As per the [documentation](#), Gradient boosting is fairly robust (*as compared to AdaBoost*) to over-fitting (why?) so a large number usually results in better performance. Note that the number of trees still need to be tuned for optimal performance.

```

def get_models():
    models = dict()
    # define number of trees to consider
    n_trees = [2, 5, 10, 50, 100, 500, 1000, 2000, 5000]
    for n in n_trees:
        models[str(n)] = GradientBoostingRegressor(n_estimators=n, random_state=1, loss='huber')
    return models

# evaluate a given model using cross-validation
def evaluate_model(model, X, y):
    # define the evaluation procedure
    cv = KFold(n_splits=5, shuffle=True, random_state=1)
    # evaluate the model and collect the results
    scores = np.sqrt(-cross_val_score(model, X, y, scoring='neg_mean_squared_error', cv=cv, n_jobs=-1))
    return scores

# get the models to evaluate
models = get_models()
# evaluate the models and store results
results, names = list(), list()
for name, model in models.items():
    # evaluate the model
    scores = evaluate_model(model, X, y)
    # store the results
    results.append(scores)
    names.append(name)
    # summarize the performance along the way
    print('>%s %.3f (%.3f)' % (name, np.mean(scores), np.std(scores)))

# plot model performance for comparison
plt.boxplot(results, labels=names, showmeans=True)
plt.ylabel('Cross validation error', fontsize=15)
plt.xlabel('Number of trees', fontsize=15)

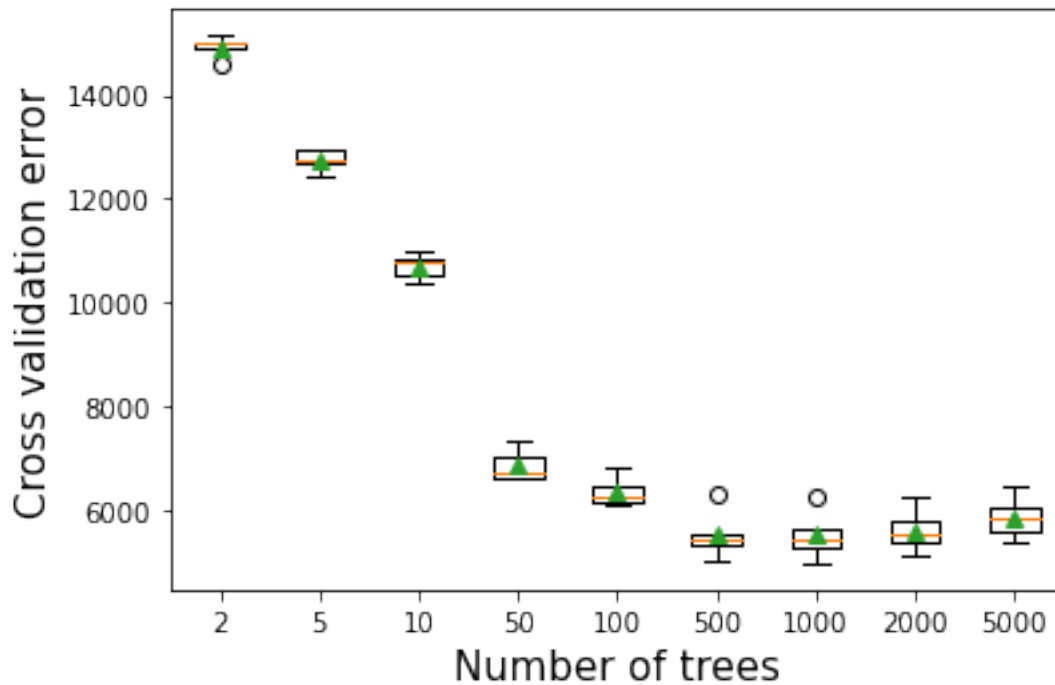
```

```

>2 14927.566 (179.475)
>5 12743.148 (189.408)
>10 10704.199 (226.234)
>50 6869.066 (278.885)
>100 6354.656 (270.097)
>500 5515.622 (424.516)
>1000 5515.251 (427.767)
>2000 5600.041 (389.687)
>5000 5854.168 (362.223)

```

```
Text(0.5, 0, 'Number of trees')
```



### 10.2.2 Depth of tree vs cross validation error

As the depth of each weak learner (decision tree) increases, the complexity of the weak learner will increase. As the complexity increases, the prediction bias will decrease, while the prediction variance will increase. Thus, there will be an optimal depth of each weak learner that minimizes the prediction error.

```
# get a list of models to evaluate
def get_models():
    models = dict()
    # explore depths from 1 to 10
    for i in range(1,21):
        # define ensemble model
        models[str(i)] = GradientBoostingRegressor(n_estimators=50,random_state=1,max_depth=i)
    return models

# evaluate a given model using cross-validation
def evaluate_model(model, X, y):
    # define the evaluation procedure
```

```

cv = KFold(n_splits=10, shuffle=True, random_state=1)
# evaluate the model and collect the results
scores = np.sqrt(-cross_val_score(model, X, y, scoring='neg_mean_squared_error', cv=cv, n_jobs=-1))
return scores

# get the models to evaluate
models = get_models()
# evaluate the models and store results
results, names = list(), list()
for name, model in models.items():
    # evaluate the model
    scores = evaluate_model(model, X, y)
    # store the results
    results.append(scores)
    names.append(name)
    # summarize the performance along the way
    print('>%s %.3f (%.3f)' % (name, np.mean(scores), np.std(scores)))
plt.boxplot(results, labels=names, showmeans=True)
plt.ylabel('Cross validation error',fontsize=15)
plt.xlabel('Depth of each tree',fontsize=15)

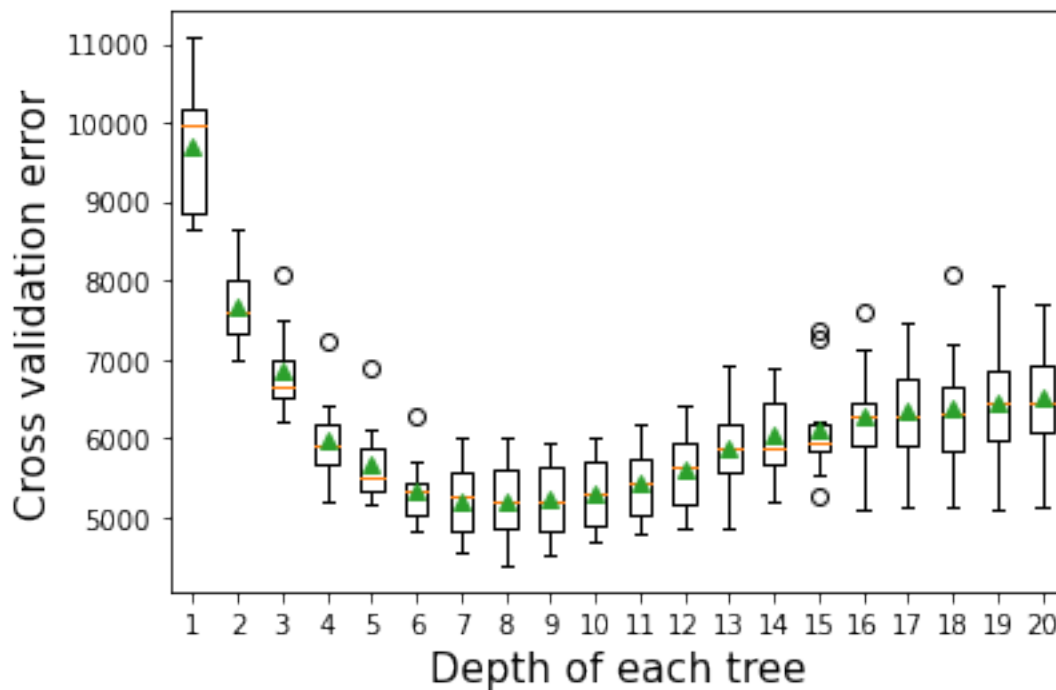
```

```

>1 9693.731 (810.090)
>2 7682.569 (489.841)
>3 6844.225 (536.792)
>4 5972.203 (538.693)
>5 5664.563 (497.882)
>6 5329.130 (404.330)
>7 5210.934 (461.038)
>8 5197.204 (494.957)
>9 5227.975 (478.789)
>10 5299.782 (446.509)
>11 5433.822 (451.673)
>12 5617.946 (509.797)
>13 5876.424 (542.981)
>14 6030.507 (560.447)
>15 6125.914 (643.852)
>16 6294.784 (672.646)
>17 6342.327 (677.050)
>18 6372.418 (791.068)
>19 6456.471 (741.693)
>20 6503.622 (759.193)

```

```
Text(0.5, 0, 'Depth of each tree')
```



### 10.2.3 Learning rate vs cross validation error

The optimal learning rate will depend on the number of trees, and vice-versa. If the learning rate is too low, it will take several trees to “learn” the response. If the learning rate is high, the response will be “learned” quickly (with fewer) trees. Learning too quickly will be prone to overfitting, while learning too slowly will be computationally expensive. Thus, there will be an optimal learning rate to minimize the prediction error.

```
def get_models():
    models = dict()
    # explore learning rates from 0.1 to 2 in 0.1 increments
    for i in np.arange(0.1, 2.1, 0.1):
        key = '%.1f' % i
        models[key] = GradientBoostingRegressor(learning_rate=i, random_state=1, loss='huber')
    return models

# evaluate a given model using cross-validation
def evaluate_model(model, X, y):
```

```

# define the evaluation procedure
cv = KFold(n_splits=10, shuffle=True, random_state=1)
# evaluate the model and collect the results
scores = np.sqrt(-cross_val_score(model, X, y, scoring='neg_mean_squared_error', cv=cv, n_jobs=-1))
return scores

# get the models to evaluate
models = get_models()
# evaluate the models and store results
results, names = list(), list()
for name, model in models.items():
    # evaluate the model
    scores = evaluate_model(model, X, y)
    # store the results
    results.append(scores)
    names.append(name)
    # summarize the performance along the way
    print('>%s %.1f (%.1f)' % (name, np.mean(scores), np.std(scores)))
# plot model performance for comparison
plt.figure(figsize=(7, 7))
plt.boxplot(results, labels=names, showmeans=True)
plt.ylabel('Cross validation error',fontsize=15)
plt.xlabel('Learning rate',fontsize=15)

```

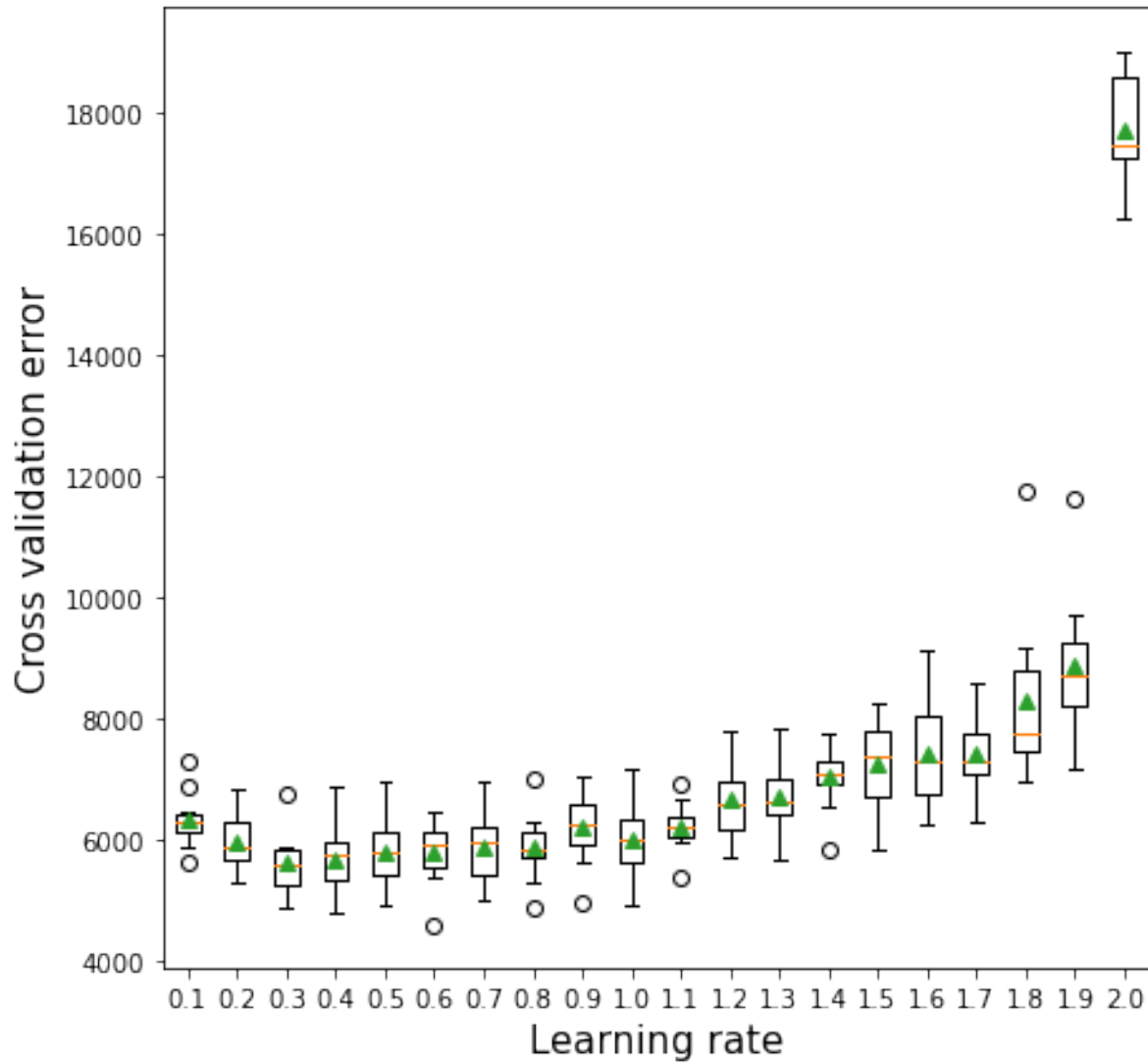
```

>0.1 6329.8 (450.7)
>0.2 5942.9 (454.8)
>0.3 5618.4 (490.8)
>0.4 5665.9 (577.3)
>0.5 5783.5 (561.7)
>0.6 5773.8 (500.3)
>0.7 5875.5 (565.7)
>0.8 5878.5 (540.5)
>0.9 6214.4 (594.3)
>1.0 5986.1 (601.5)
>1.1 6216.5 (395.3)
>1.2 6667.5 (657.2)
>1.3 6717.4 (594.4)
>1.4 7048.4 (531.7)
>1.5 7265.0 (742.0)
>1.6 7404.4 (868.2)
>1.7 7425.8 (606.3)
>1.8 8283.0 (1345.3)

```

```
>1.9 8872.2 (1137.9)  
>2.0 17713.3 (865.3)
```

```
Text(0.5, 0, 'Learning rate')
```



#### 10.2.4 Subsampling vs cross validation error

```

def get_models():
    models = dict()
    # explore learning rates from 0.1 to 2 in 0.1 increments
    for s in np.arange(0.25, 1.1, 0.25):
        key = '%.2f' % s
        models[key] = GradientBoostingRegressor(random_state=1, subsample=s, loss='huber')
    return models

# evaluate a given model using cross-validation
def evaluate_model(model, X, y):
    # define the evaluation procedure
    cv = KFold(n_splits=10, shuffle=True, random_state=1)
    # evaluate the model and collect the results
    scores = np.sqrt(-cross_val_score(model, X, y, scoring='neg_mean_squared_error', cv=cv, n_jobs=-1))
    return scores

# get the models to evaluate
models = get_models()
# evaluate the models and store results
results, names = list(), list()
for name, model in models.items():
    # evaluate the model
    scores = evaluate_model(model, X, y)
    # store the results
    results.append(scores)
    names.append(name)
    # summarize the performance along the way
    print('>%s %.2f (%.2f)' % (name, np.mean(scores), np.std(scores)))
# plot model performance for comparison
plt.figure(figsize=(7, 7))
plt.boxplot(results, labels=names, showmeans=True)
plt.ylabel('Cross validation error', fontsize=15)
plt.xlabel('Subsample', fontsize=15)

```

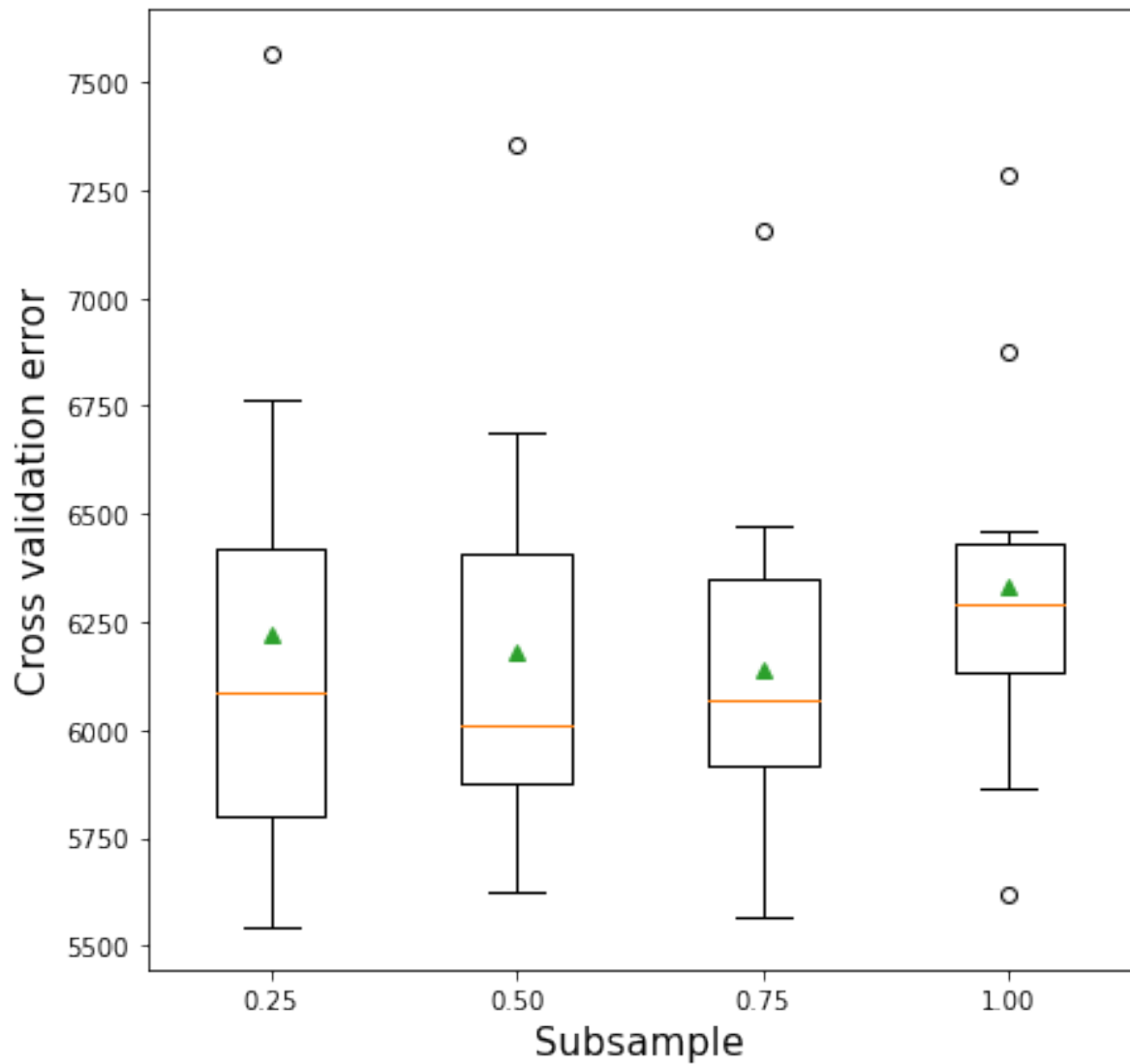
```

>0.25 6219.59 (569.97)
>0.50 6178.28 (501.87)
>0.75 6141.96 (432.66)
>1.00 6329.79 (450.72)

```

```
Text(0.5, 0, 'Subsample')
```





### 10.2.5 Maximum features vs cross-validation error

```
def get_models():
    models = dict()
    # explore learning rates from 0.1 to 2 in 0.1 increments
    for s in np.arange(0.25, 1.1, 0.25):
        key = '%.2f' % s
        models[key] = GradientBoostingRegressor(random_state=1, max_features=s, loss='huber')
    return models
```

```

# evaluate a given model using cross-validation
def evaluate_model(model, X, y):
    # define the evaluation procedure
    cv = KFold(n_splits=10, shuffle=True, random_state=1)
    # evaluate the model and collect the results
    scores = np.sqrt(-cross_val_score(model, X, y, scoring='neg_mean_squared_error', cv=cv, n_jobs=-1))
    return scores

# get the models to evaluate
models = get_models()
# evaluate the models and store results
results, names = list(), list()
for name, model in models.items():
    # evaluate the model
    scores = evaluate_model(model, X, y)
    # store the results
    results.append(scores)
    names.append(name)
    # summarize the performance along the way
    print('>%s %.2f (%.2f)' % (name, np.mean(scores), np.std(scores)))
# plot model performance for comparison
plt.figure(figsize=(7, 7))
plt.boxplot(results, labels=names, showmeans=True)
plt.ylabel('Cross validation error',fontsize=15)
plt.xlabel('Maximum features',fontsize=15)

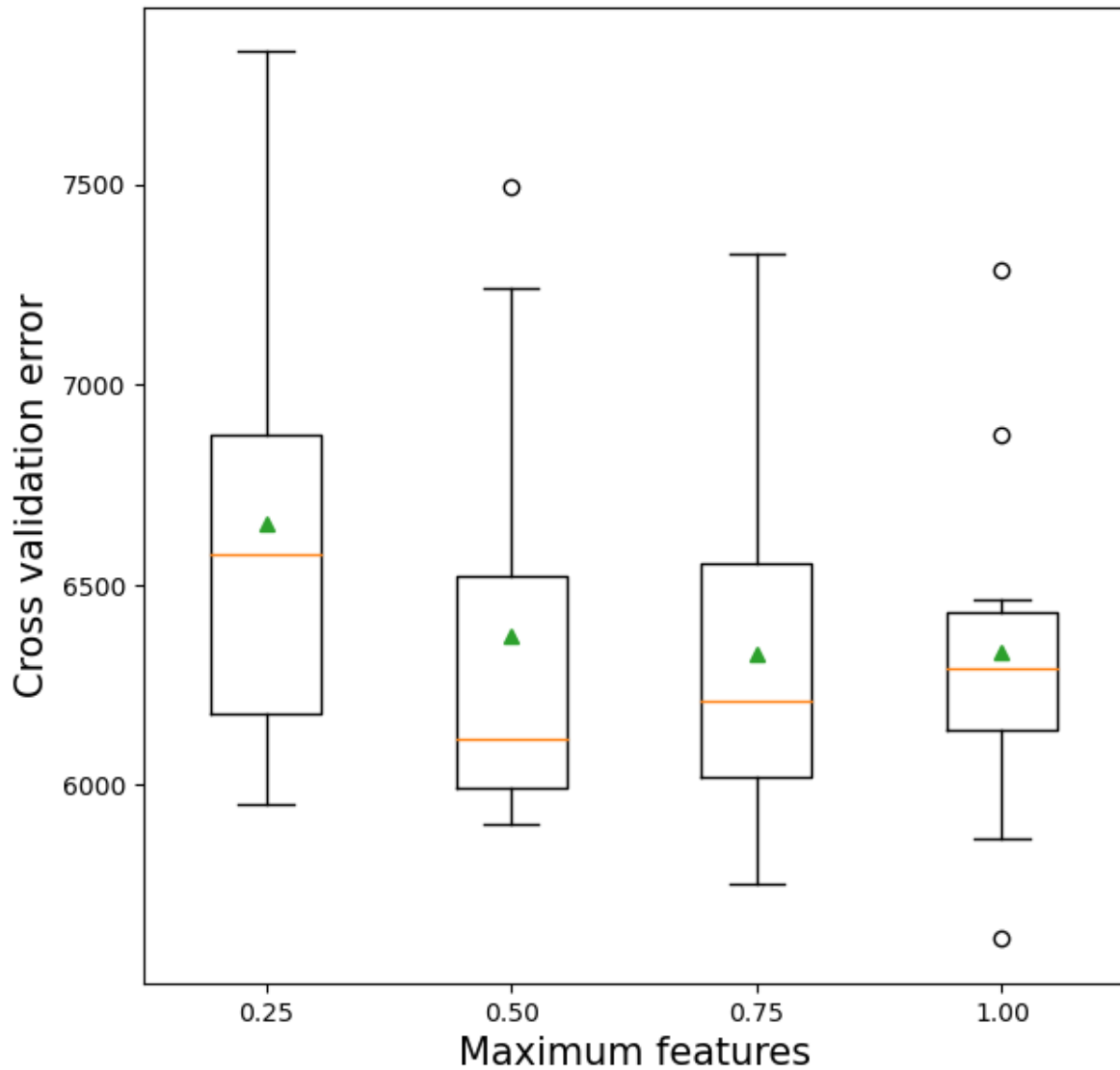
```

```

>0.25 6654.27 (567.72)
>0.50 6373.92 (538.53)
>0.75 6325.55 (470.41)
>1.00 6329.79 (450.72)

```

```
Text(0.5, 0, 'Maximum features')
```



### 10.2.6 Tuning Gradient boosting for regression

As the optimal value of the parameters depend on each other, we need to optimize them simultaneously.

```
start_time = time.time()
model = GradientBoostingRegressor(random_state=1, loss='huber')
grid = dict()
grid['n_estimators'] = [10, 50, 100, 200, 500]
```

```

grid['learning_rate'] = [0.0001, 0.001, 0.01, 0.1, 1.0]
grid['max_depth'] = [3, 5, 8, 10, 12, 15]

# define the evaluation procedure
cv = KFold(n_splits=5, shuffle=True, random_state=1)
# define the grid search procedure
grid_search = GridSearchCV(estimator=model, param_grid=grid, n_jobs=-1, cv=cv, scoring='neg_r
                           verbose = True)
# execute the grid search
grid_result = grid_search.fit(X, y)
# summarize the best score and configuration
print("Best: %f using %s" % (np.sqrt(-grid_result.best_score_), grid_result.best_params_))
# summarize all scores that were evaluated
means = grid_result.cv_results_['mean_test_score']
stds = grid_result.cv_results_['std_test_score']
params = grid_result.cv_results_['params']
# for mean, stdev, param in zip(means, stds, params):
#     print("%f (%f) with: %r" % (mean, stdev, param))
print("Time taken = ", (time.time()-start_time)/60, " minutes")

```

Best: 5190.765919 using {'learning\_rate': 0.1, 'max\_depth': 8, 'n\_estimators': 100}  
Time taken = 46.925597019990285 minutes

Note that the code takes 46 minutes to run. In case of a lot of hyperparameters, [RandomizedSearchCV](#) may be preferred to trade-off between optimality of the solution and computational cost.

```

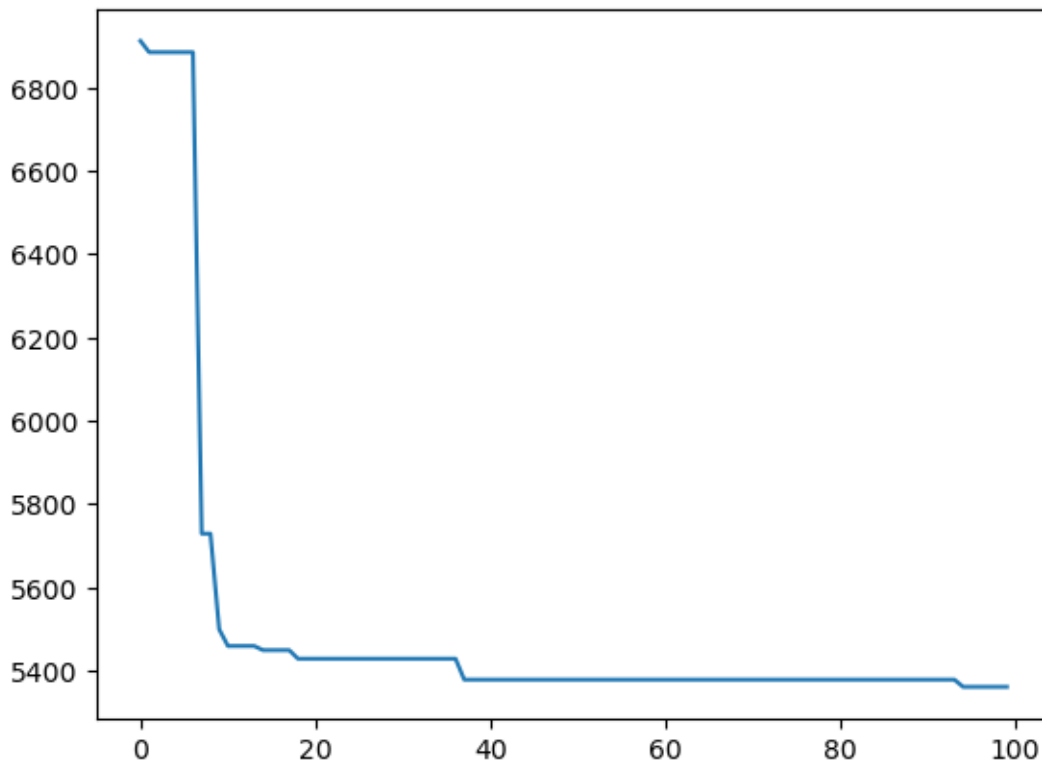
model = GradientBoostingRegressor(random_state=1, loss='huber')
grid = dict()
grid['n_estimators'] = Integer(2, 1000)
grid['learning_rate'] = Real(0.0001, 1.0)
grid['max_leaf_nodes'] = Integer(4, 5000)
grid['subsample'] = Real(0.1, 1)
grid['max_features'] = Real(0.1, 1)

kfold = KFold(n_splits = 5, shuffle = True, random_state = 1)
gcv = BayesSearchCV(model, search_spaces = grid, cv = kfold, n_iter = 100, random_state = 1,
                    scoring = 'neg_root_mean_squared_error', n_jobs = -1)
paras = list(gcv.search_spaces.keys())
paras.sort()
start_time = time.time()

```

```
def monitor(optim_result):
    cv_values = pd.Series(optim_result['func_vals']).cummin()
    display.clear_output(wait = True)
    min_ind = pd.Series(optim_result['func_vals']).argmin()
    print(paras, "=", optim_result['x_iters'][min_ind], pd.Series(optim_result['func_vals']))
    print("Time so far = ", np.round((time.time()-start_time)/60), "minutes")
    sns.lineplot(cv_values)
    plt.show()
gcv.fit(X, y, callback = monitor)
```

```
['learning_rate', 'max_features', 'max_leaf_nodes', 'n_estimators', 'subsample'] = [0.231020
Time so far = 21.0 minutes
```



```
BayesSearchCV(cv=KFold(n_splits=5, random_state=1, shuffle=True),
               estimator=GradientBoostingRegressor(loss='huber', random_state=1),
               n_iter=100, n_jobs=-1, random_state=1,
               scoring='neg_root_mean_squared_error',
               search_spaces={'learning_rate': Real(low=0.0001, high=1.0, prior='uniform', tr
```

```
'max_features': Real(low=0.1, high=1, prior='uniform', transform='log'),
'max_leaf_nodes': Integer(low=4, high=5000, prior='uniform', transform='log'),
'n_estimators': Integer(low=2, high=1000, prior='uniform', transform='log'),
'subsample': Real(low=0.1, high=1, prior='uniform', transform='log')
```

```
#Model based on the optimal parameters
model = GradientBoostingRegressor(max_depth=8,n_estimators=100,learning_rate=0.1,
                                  random_state=1,loss='huber').fit(X,y)
```

```
#RMSE of the optimized model on test data
print("Gradient boost RMSE = ",np.sqrt(mean_squared_error(model.predict(Xtest),ytest)))
```

Gradient boost RMSE = 5405.787029062213

```
#Model based on the optimal parameters
model_bayes = GradientBoostingRegressor(max_leaf_nodes=5000,n_estimators=817,learning_rate=0.1,
                                       random_state=1,subsample=1.0,loss='huber').fit(X,y)
```

```
#RMSE of the optimized model on test data
print("Gradient boost RMSE = ",np.sqrt(mean_squared_error(model_bayes.predict(Xtest),ytest)))
```

Gradient boost RMSE = 5734.200307094321

```
#Let us combine the Gradient boost model with other models
model2 = AdaBoostRegressor(base_estimator=DecisionTreeRegressor(max_depth=10),n_estimators=500,
                           random_state=1).fit(X,y)
print("AdaBoost RMSE = ",np.sqrt(mean_squared_error(model2.predict(Xtest),ytest)))
model3 = RandomForestRegressor(n_estimators=300, random_state=1,
                              n_jobs=-1, max_features=2).fit(X, y)
print("Random Forest RMSE = ",np.sqrt(mean_squared_error(model3.predict(Xtest),ytest)))
```

AdaBoost RMSE = 5693.165811600585

Random Forest RMSE = 5642.45839697972

```
#Ensemble model
pred1=model.predict(Xtest)#Gradient boost
pred2=model2.predict(Xtest)#Adaboost
pred3=model3.predict(Xtest)#Random forest
pred = 0.34*pred1+0.33*pred2+0.33*pred3 #Higher weight to the better model
print("Ensemble model RMSE = ", np.sqrt(mean_squared_error(pred,ytest)))
```

Ensemble model RMSE = 5364.478227748279

### 10.2.7 Ensemble modeling (for regression models)

```
#Ensemble model
pred1=model.predict(Xtest)#Gradient boost
pred2=model2.predict(Xtest)#Adaboost
pred3=model3.predict(Xtest)#Random forest
pred = 0.6*pred1+0.2*pred2+0.2*pred3 #Higher weight to the better model
print("Ensemble model RMSE = ", np.sqrt(mean_squared_error(pred,ytest)))
```

Ensemble model RMSE = 5323.119083375402

Combined, the random forest model, gradient boost and the Adaboost model do better than each of the individual models.

Note that ideally we should do K-fold cross validation to figure out the optimal weights. We'll learn about ensembling techniques later in the course.

## 10.3 Gradient boosting for classification

Below is the Gradient boost implementation on a classification problem. The takeaways are the same as that of the regression problem above.

```
train = pd.read_csv('./Datasets/diabetes_train.csv')
test = pd.read_csv('./Datasets/diabetes_test.csv')
```

```
X = train.drop(columns = 'Outcome')
Xtest = test.drop(columns = 'Outcome')
y = train['Outcome']
ytest = test['Outcome']
```

### 10.3.1 Number of trees vs cross validation accuracy

```
def get_models():
    models = dict()
    # define number of trees to consider
    n_trees = [10, 50, 100, 500, 1000, 5000]
    for n in n_trees:
```

```

        models[str(n)] = GradientBoostingClassifier(n_estimators=n,random_state=1)
    return models

# evaluate a given model using cross-validation
def evaluate_model(model, X, y):
    # define the evaluation procedure
    cv = StratifiedKFold(n_splits=10, shuffle=True, random_state=1)
    # evaluate the model and collect the results
    scores = cross_val_score(model, X, y, scoring='accuracy', cv=cv, n_jobs=-1)
    return scores

# get the models to evaluate
models = get_models()
# evaluate the models and store results
results, names = list(), list()
for name, model in models.items():
    # evaluate the model
    scores = evaluate_model(model, X, y)
    # store the results
    results.append(scores)
    names.append(name)
    # summarize the performance along the way
    print('>%s %.3f (%.3f)' % (name, np.mean(scores), np.std(scores)))
# plot model performance for comparison
plt.boxplot(results, labels=names, showmeans=True)
plt.ylabel('Cross validation error',fontsize=15)
plt.xlabel('Number of trees',fontsize=15)

```

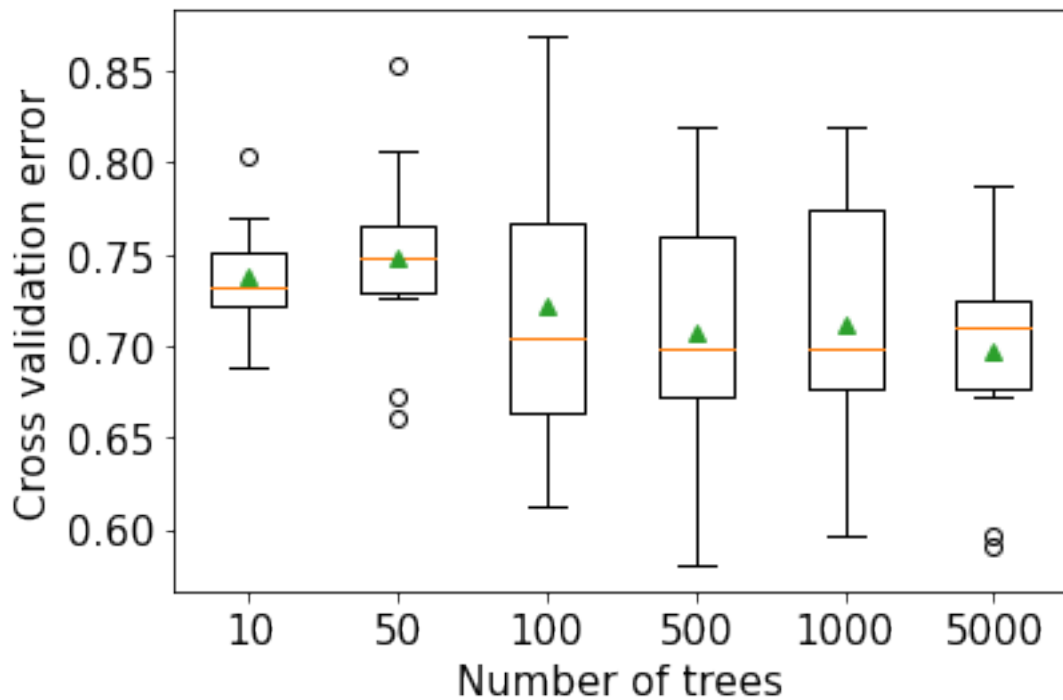
```

>10 0.738 (0.031)
>50 0.748 (0.054)
>100 0.722 (0.075)
>500 0.707 (0.066)
>1000 0.712 (0.075)
>5000 0.697 (0.061)

```

```
Text(0.5, 0, 'Number of trees')
```





### 10.3.2 Depth of each tree vs cross validation accuracy

```
# get a list of models to evaluate
def get_models():
    models = dict()
    # explore depths from 1 to 10
    for i in range(1,21):
        # define ensemble model
        models[str(i)] = GradientBoostingClassifier(random_state=1,max_depth=i)
    return models

# evaluate a given model using cross-validation
def evaluate_model(model, X, y):
    # define the evaluation procedure
    cv = StratifiedKFold(n_splits=10, shuffle=True, random_state=1)
    # evaluate the model and collect the results
    scores = cross_val_score(model, X, y, scoring='accuracy', cv=cv, n_jobs=-1)
    return scores

# get the models to evaluate
```

```

models = get_models()
# evaluate the models and store results
results, names = list(), list()
for name, model in models.items():
    # evaluate the model
    scores = evaluate_model(model, X, y)
    # store the results
    results.append(scores)
    names.append(name)
    # summarize the performance along the way
    print('>%s %.3f (%.3f)' % (name, np.mean(scores), np.std(scores)))
plt.boxplot(results, labels=names, showmeans=True)
plt.ylabel('Accuracy',fontsize=15)
plt.xlabel('Depth of each tree',fontsize=15)

```

```

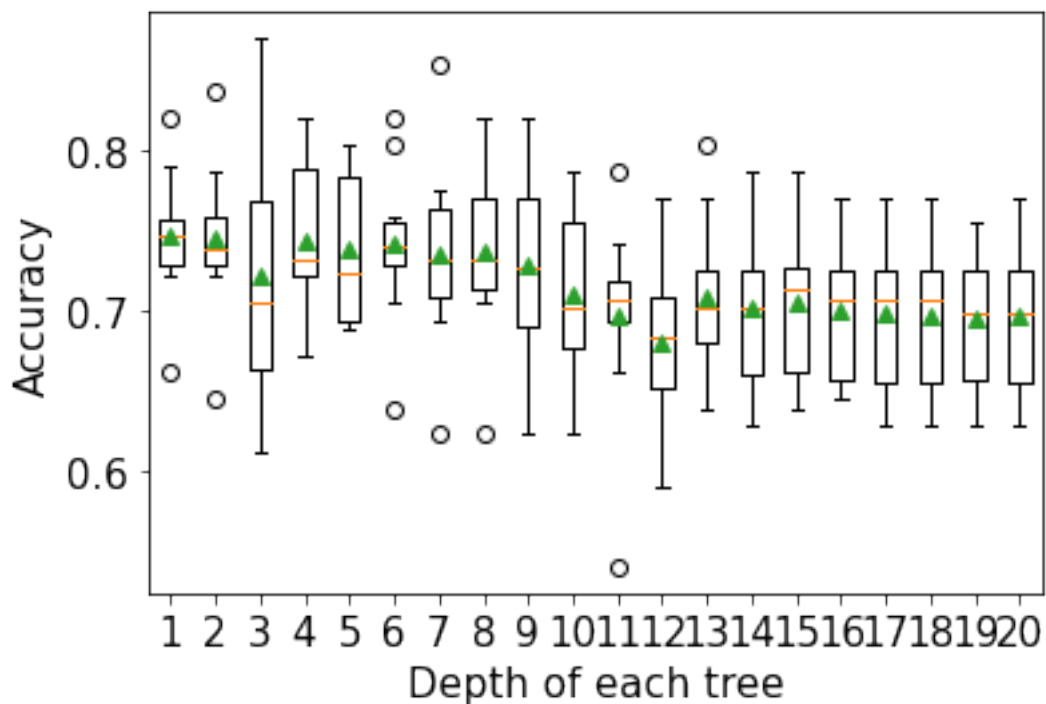
>1 0.746 (0.040)
>2 0.744 (0.046)
>3 0.722 (0.075)
>4 0.743 (0.049)
>5 0.738 (0.046)
>6 0.741 (0.047)
>7 0.735 (0.057)
>8 0.736 (0.051)
>9 0.728 (0.055)
>10 0.710 (0.050)
>11 0.697 (0.061)
>12 0.681 (0.056)
>13 0.709 (0.047)
>14 0.702 (0.048)
>15 0.705 (0.048)
>16 0.700 (0.042)
>17 0.699 (0.048)
>18 0.697 (0.050)
>19 0.696 (0.042)
>20 0.697 (0.048)

```

```

Text(0.5, 0, 'Depth of each tree')

```



### 10.3.3 Learning rate vs cross validation accuracy

```
def get_models():
    models = dict()
    # explore learning rates from 0.1 to 2 in 0.1 increments
    for i in np.arange(0.1, 2.1, 0.1):
        key = '%.1f' % i
        models[key] = GradientBoostingClassifier(learning_rate=i, random_state=1)
    return models

# evaluate a given model using cross-validation
def evaluate_model(model, X, y):
    # define the evaluation procedure
    cv = KFold(n_splits=10, shuffle=True, random_state=1)
    # evaluate the model and collect the results
    scores = cross_val_score(model, X, y, scoring='accuracy', cv=cv, n_jobs=-1)
    return scores

# get the models to evaluate
models = get_models()
```

```

# evaluate the models and store results
results, names = list(), list()
for name, model in models.items():
    # evaluate the model
    scores = evaluate_model(model, X, y)
    # store the results
    results.append(scores)
    names.append(name)
    # summarize the performance along the way
    print('>%s %.3f (%.3f)' % (name, np.mean(scores), np.std(scores)))
# plot model performance for comparison
plt.figure(figsize=(7, 7))
plt.boxplot(results, labels=names, showmeans=True)
plt.ylabel('Accuracy',fontsize=15)
plt.xlabel('Learning rate',fontsize=15)

```

```

>0.1 0.747 (0.044)
>0.2 0.736 (0.028)
>0.3 0.726 (0.039)
>0.4 0.730 (0.034)
>0.5 0.726 (0.041)
>0.6 0.722 (0.043)
>0.7 0.717 (0.050)
>0.8 0.713 (0.033)
>0.9 0.694 (0.045)
>1.0 0.695 (0.032)
>1.1 0.718 (0.034)
>1.2 0.692 (0.045)
>1.3 0.708 (0.042)
>1.4 0.704 (0.050)
>1.5 0.702 (0.028)
>1.6 0.700 (0.050)
>1.7 0.694 (0.044)
>1.8 0.650 (0.075)
>1.9 0.551 (0.163)
>2.0 0.484 (0.123)

```

```
Text(0.5, 0, 'Learning rate')
```



### 10.3.4 Tuning Gradient boosting Classifier

```
start_time = time.time()
model = GradientBoostingClassifier(random_state=1)
grid = dict()
grid['n_estimators'] = [10, 50, 100, 200, 500]
grid['learning_rate'] = [0.0001, 0.001, 0.01, 0.1, 1.0]
grid['max_depth'] = [1, 2, 3, 4, 5]
```

```

grid['subsample'] = [0.5,1.0]
# define the evaluation procedure
cv = StratifiedKFold(n_splits=5, shuffle=True, random_state=1)
# define the grid search procedure
grid_search = GridSearchCV(estimator=model, param_grid=grid, n_jobs=-1, cv=cv, verbose = True)
# execute the grid search
grid_result = grid_search.fit(X, y)
# summarize the best score and configuration
print("Best: %f using %s" % (grid_result.best_score_, grid_result.best_params_))
print("Time taken = ", time.time() - start_time, "seconds")

```

Fitting 5 folds for each of 250 candidates, totalling 1250 fits

Best: 0.701045 using {'learning\_rate': 1.0, 'max\_depth': 3, 'n\_estimators': 200, 'subsample': 0.5}

Time taken = 32.46394085884094

```

#Model based on the optimal parameters
model = GradientBoostingClassifier(random_state=1,max_depth=3,learning_rate=0.1,subsample=0.5,
                                   n_estimators=200).fit(X,y)

# Note that we are using the cross-validated predicted probabilities, instead of directly using
# predicted probabilities on train data, as the model may be overfitting on the train data, and
# may lead to misleading results
cross_val_ypred = cross_val_predict(GradientBoostingClassifier(random_state=1,max_depth=3,
                                                                learning_rate=0.1,subsample=0.5,
                                                                n_estimators=200), X, y, cv = 5, method = 'predict_proba')

p, r, thresholds = precision_recall_curve(y, cross_val_ypred[:,1])
def plot_precision_recall_vs_threshold(precisions, recalls, thresholds):
    plt.figure(figsize=(8, 8))
    plt.title("Precision and Recall Scores as a function of the decision threshold")
    plt.plot(thresholds, precisions[:-1], "b--", label="Precision")
    plt.plot(thresholds, recalls[:-1], "g-", label="Recall")
    plt.plot(thresholds, precisions[:-1], "o", color = 'blue')
    plt.plot(thresholds, recalls[:-1], "o", color = 'green')
    plt.ylabel("Score")
    plt.xlabel("Decision Threshold")
    plt.legend(loc='best')
    plt.legend()
plot_precision_recall_vs_threshold(p, r, thresholds)

```



```
# Thresholds with precision and recall
all_thresholds = np.concatenate([thresholds.reshape(-1,1), p[:-1].reshape(-1,1), r[:-1].reshape(-1,1)])
recall_more_than_80 = all_thresholds[all_thresholds[:,2]>0.8,:]
# As the values in 'recall_more_than_80' are arranged in decreasing order of recall and increasing order of precision,
# the last value will provide the maximum threshold probability for the recall to be more than 80%
# We wish to find the maximum threshold probability to obtain the maximum possible precision
recall_more_than_80[recall_more_than_80.shape[0]-1]
```

```
array([0.18497144, 0.53205128, 0.80193237])
```

```
#Optimal decision threshold probability
thres = recall_more_than_80[recall_more_than_80.shape[0]-1][0]
thres
```

```
0.18497143500912738
```

```
# Performance metrics computation for the optimum decision threshold probability
desired_threshold = thres

y_pred_prob = model.predict_proba(Xtest)[:,-1]

# Classifying observations in the positive class (y = 1) if the predicted probability is greater
# than the desired decision threshold probability
y_pred = y_pred_prob > desired_threshold
y_pred = y_pred.astype(int)

#Computing the accuracy
print("Accuracy: ",accuracy_score(y_pred, ytest)*100)

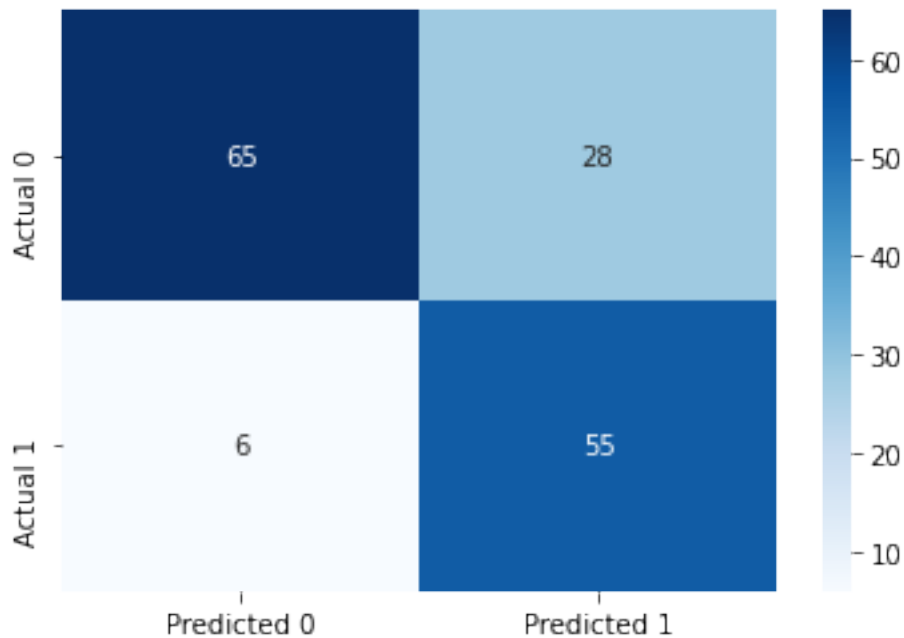
#Computing the ROC-AUC
fpr, tpr, auc_thresholds = roc_curve(ytest, y_pred_prob)
print("ROC-AUC: ",auc(fpr, tpr))# AUC of ROC

#Computing the precision and recall
print("Precision: ", precision_score(ytest, y_pred))
print("Recall: ", recall_score(ytest, y_pred))

#Confusion matrix
cm = pd.DataFrame(confusion_matrix(ytest, y_pred),
                  columns=['Predicted 0', 'Predicted 1'], index = ['Actual 0', 'Actual 1'])
sns.heatmap(cm, annot=True, cmap='Blues', fmt='g');
```

```
Accuracy: 77.92207792207793
ROC-AUC: 0.8704389212057112
Precision: 0.6626506024096386
Recall: 0.9016393442622951
```





The model seems to be similar to the Adaboost model. However, gradient boosting algorithms with robust loss functions can perform better than Adaboost in the presence of outliers (*in terms of response*) in the data.

## 10.4 Faster algorithms and tuning tips

Check out `HistGradientBoostingRegressor()` and `HistGradientBoostingClassifier()` for a faster gradient boosting algorithm for big datasets (*more than 10,000 observations*).

Check out tips for faster hyperparameter tuning, such as tuning `max_leaf_nodes` instead of `max_depth` [here](#).

# 11 XGBoost

XGBoost is a very recently developed algorithm (2016). Thus, it's not yet there in standard textbooks. Here are some resources for it.

[Documentation](#)

[Slides](#)

[Reference paper](#)

[Video by author \(Tianqi Chen\)](#)

[Video by StatQuest](#)

## 11.1 Hyperparameters

The following are some of the important hyperparameters to tune in XGBoost:

1. Number of trees (`n_estimators`)
2. Depth of each tree (`max_depth`)
3. Learning rate (`learning_rate`)
4. Sampling observations / predictors (`subsample` for observations, `colsample_bytree` for predictors)
5. Regularization parameters (`reg_lambda` & `gamma`)

However, there are other hyperparameters that can be tuned as well. Check out the list of all hyperparameters in the XGBoost [documentation](#).

```
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import cross_val_score, train_test_split, KFold, cross_val_predict
from sklearn.metrics import mean_squared_error, r2_score, roc_curve, auc, precision_recall_curve
```

```

recall_score, precision_score, confusion_matrix
from sklearn.tree import DecisionTreeRegressor, DecisionTreeClassifier
from sklearn.model_selection import GridSearchCV, ParameterGrid, StratifiedKFold, RandomizedSearchCV
from sklearn.ensemble import VotingRegressor, VotingClassifier, StackingRegressor, StackingClassifier
from sklearn.linear_model import LinearRegression, LogisticRegression, LassoCV, RidgeCV, ElasticNetCV
from sklearn.neighbors import KNeighborsRegressor
import itertools as it
import time as time
import xgboost as xgb
from skopt import BayesSearchCV
from skopt.space import Real, Categorical, Integer
from skopt.plots import plot_objective, plot_histogram, plot_convergence
import warnings
from IPython import display

```

```

#Using the same datasets as used for linear regression in STAT303-2,
#so that we can compare the non-linear models with linear regression
trainf = pd.read_csv('./Datasets/Car_features_train.csv')
trainp = pd.read_csv('./Datasets/Car_prices_train.csv')
testf = pd.read_csv('./Datasets/Car_features_test.csv')
testp = pd.read_csv('./Datasets/Car_prices_test.csv')
train = pd.merge(trainf, trainp)
test = pd.merge(testf, testp)
train.head()

```

	carID	brand	model	year	transmission	mileage	fuelType	tax	mpg	engineSize	price
0	18473	bmw	6 Series	2020	Semi-Auto	11	Diesel	145	53.3282	3.0	37980
1	15064	bmw	6 Series	2019	Semi-Auto	10813	Diesel	145	53.0430	3.0	33980
2	18268	bmw	6 Series	2020	Semi-Auto	6	Diesel	145	53.4379	3.0	36850
3	18480	bmw	6 Series	2017	Semi-Auto	18895	Diesel	145	51.5140	3.0	25998
4	18492	bmw	6 Series	2015	Automatic	62953	Diesel	160	51.4903	3.0	18990

```

X = train[['mileage', 'mpg', 'year', 'engineSize']]
Xtest = test[['mileage', 'mpg', 'year', 'engineSize']]
y = train['price']
ytest = test['price']

```

## 11.2 XGBoost for regression

### 11.2.1 Number of trees vs cross validation error

As the number of trees increase, the prediction bias will decrease. Like gradient boosting is relatively robust (*as compared to AdaBoost*) to over-fitting (why?) so a large number usually results in better performance. Note that the number of trees still need to be tuned for optimal performance.

```
def get_models():
    models = dict()
    # define number of trees to consider
    n_trees = [5, 10, 50, 100, 500, 1000, 2000, 5000]
    for n in n_trees:
        models[str(n)] = xgb.XGBRegressor(n_estimators=n, random_state=1)
    return models

# evaluate a given model using cross-validation
def evaluate_model(model, X, y):
    # define the evaluation procedure
    cv = KFold(n_splits=5, shuffle=True, random_state=1)
    # evaluate the model and collect the results
    scores = np.sqrt(-cross_val_score(model, X, y, scoring='neg_mean_squared_error', cv=cv, n_jobs=-1))
    return scores

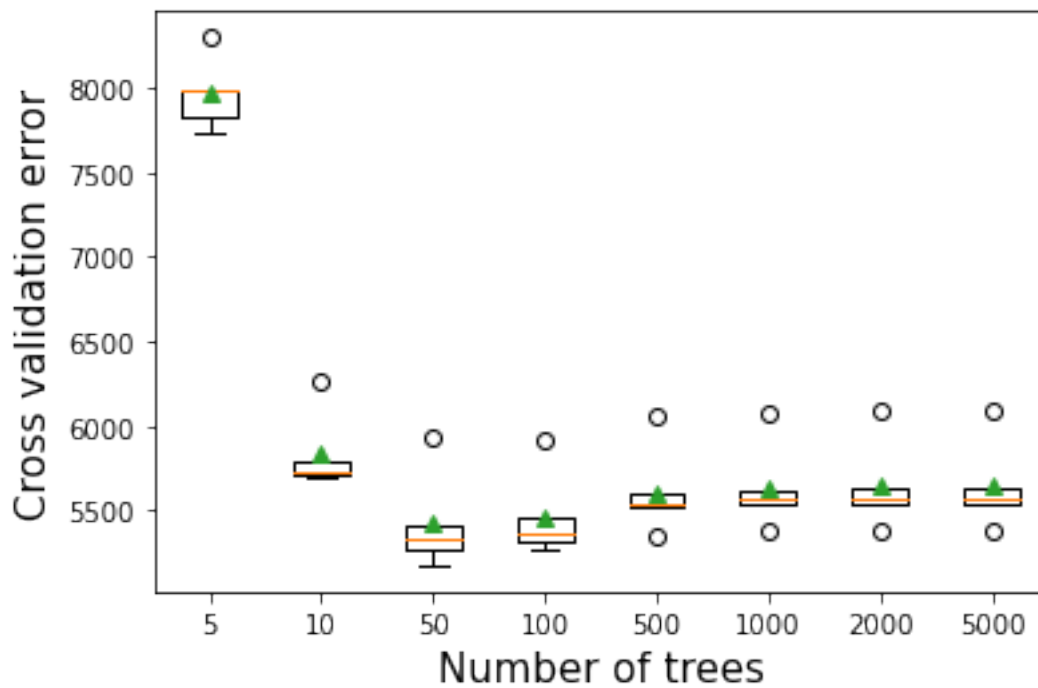
# get the models to evaluate
models = get_models()
# evaluate the models and store results
results, names = list(), list()
for name, model in models.items():
    # evaluate the model
    scores = evaluate_model(model, X, y)
    # store the results
    results.append(scores)
    names.append(name)
    # summarize the performance along the way
    print('>%s %.3f (%.3f)' % (name, np.mean(scores), np.std(scores)))
# plot model performance for comparison
plt.boxplot(results, labels=names, showmeans=True)
plt.ylabel('Cross validation error', fontsize=15)
plt.xlabel('Number of trees', fontsize=15)
```

```

>5 7961.485 (192.906)
>10 5837.134 (217.986)
>50 5424.788 (263.890)
>100 5465.396 (237.938)
>500 5608.350 (235.903)
>1000 5635.159 (236.664)
>2000 5642.669 (236.192)
>5000 5643.411 (236.074)

```

```
Text(0.5, 0, 'Number of trees')
```



### 11.2.2 Depth of tree vs cross validation error

As the depth of each weak learner (decision tree) increases, the complexity of the weak learner will increase. As the complexity increases, the prediction bias will decrease, while the prediction variance will increase. Thus, there will be an optimal depth of each weak learner that minimizes the prediction error.

```

# get a list of models to evaluate
def get_models():
    models = dict()
    # explore depths from 1 to 10
    for i in range(1,21):
        # define ensemble model
        models[str(i)] = xgb.XGBRegressor(random_state=1,max_depth=i)
    return models

# evaluate a given model using cross-validation
def evaluate_model(model, X, y):
    # define the evaluation procedure
    cv = KFold(n_splits=10, shuffle=True, random_state=1)
    # evaluate the model and collect the results
    scores = np.sqrt(-cross_val_score(model, X, y, scoring='neg_mean_squared_error', cv=cv, n_jobs=-1))
    return scores

# get the models to evaluate
models = get_models()
# evaluate the models and store results
results, names = list(), list()
for name, model in models.items():
    # evaluate the model
    scores = evaluate_model(model, X, y)
    # store the results
    results.append(scores)
    names.append(name)
    # summarize the performance along the way
    print('>%s %.3f (%.3f)' % (name, np.mean(scores), np.std(scores)))
plt.boxplot(results, labels=names, showmeans=True)
plt.ylabel('Cross validation error',fontsize=15)
plt.xlabel('Depth of each tree',fontsize=15)

```

```

>1 7541.827 (545.951)
>2 6129.425 (393.357)
>3 5647.783 (454.318)
>4 5438.481 (453.726)
>5 5358.074 (379.431)
>6 5281.675 (383.848)
>7 5495.163 (459.356)
>8 5399.145 (380.437)
>9 5469.563 (384.004)

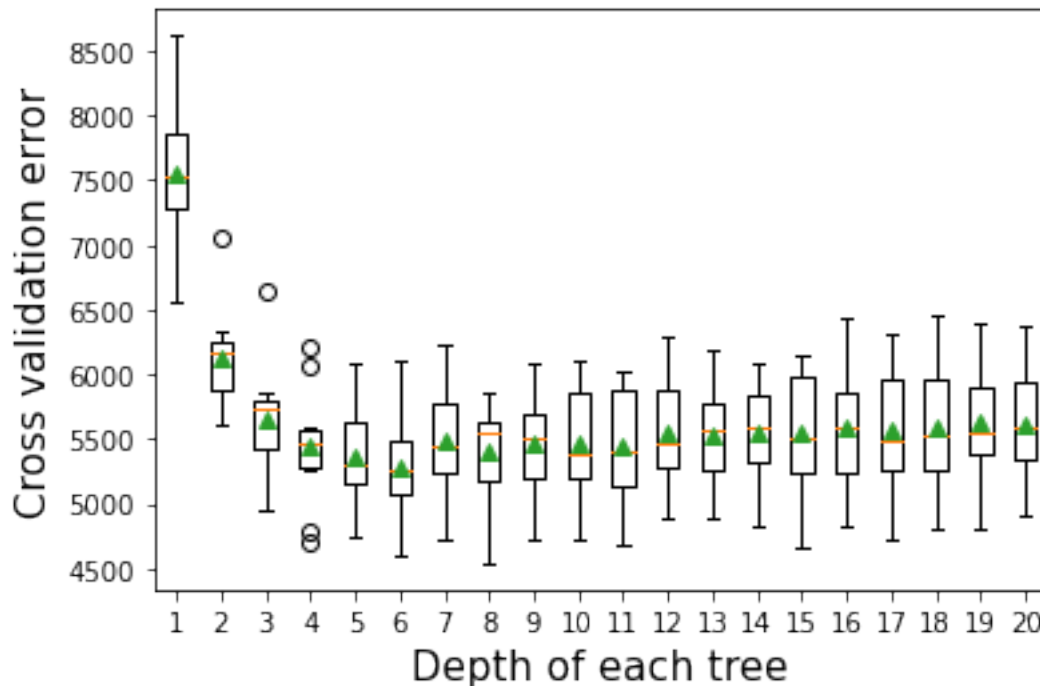
```

```

>10 5461.549 (416.630)
>11 5443.210 (432.863)
>12 5546.447 (412.097)
>13 5532.414 (369.131)
>14 5556.761 (362.746)
>15 5540.366 (452.612)
>16 5586.004 (451.199)
>17 5563.137 (464.344)
>18 5594.919 (480.221)
>19 5641.226 (451.713)
>20 5616.462 (417.405)

```

```
Text(0.5, 0, 'Depth of each tree')
```



### 11.2.3 Learning rate vs cross validation error

The optimal learning rate will depend on the number of trees, and vice-versa. If the learning rate is too low, it will take several trees to “learn” the response. If the learning rate is high, the response will be “learned” quickly (with fewer) trees. Learning too quickly will be prone to overfitting, while learning too slowly will be computationally expensive. Thus, there will be an optimal learning rate to minimize the prediction error.

```

def get_models():
    models = dict()
    # explore learning rates from 0.1 to 2 in 0.1 increments
    for i in [0.01,0.05,0.1,0.2,0.3,0.4,0.5,0.6,0.8,1.0]:
        key = '%.4f' % i
        models[key] = xgb.XGBRegressor(learning_rate=i,random_state=1)
    return models

# evaluate a given model using cross-validation
def evaluate_model(model, X, y):
    # define the evaluation procedure
    cv = KFold(n_splits=10, shuffle=True, random_state=1)
    # evaluate the model and collect the results
    scores = np.sqrt(-cross_val_score(model, X, y, scoring='neg_mean_squared_error', cv=cv, n_jobs=-1))
    return scores

# get the models to evaluate
models = get_models()
# evaluate the models and store results
results, names = list(), list()
for name, model in models.items():
    # evaluate the model
    scores = evaluate_model(model, X, y)
    # store the results
    results.append(scores)
    names.append(name)
    # summarize the performance along the way
    print('>%s %.1f (%.1f)' % (name, np.mean(scores), np.std(scores)))
# plot model performance for comparison
plt.figure(figsize=(7, 7))
plt.boxplot(results, labels=names, showmeans=True)
plt.ylabel('Cross validation error',fontsize=15)
plt.xlabel('Learning rate',fontsize=15)

```

```

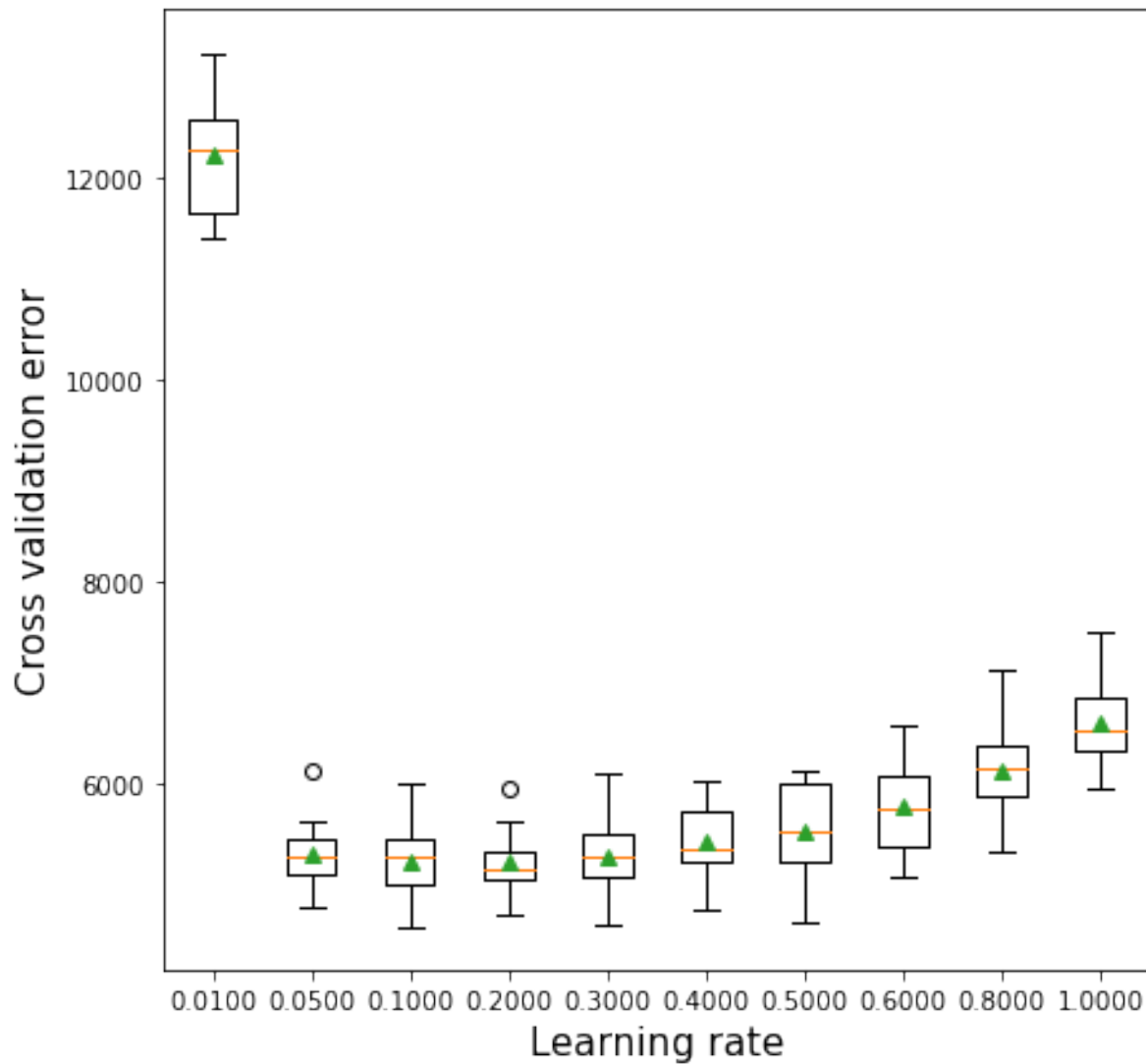
>0.0100 12223.8 (636.7)
>0.0500 5298.5 (383.5)
>0.1000 5236.3 (397.5)
>0.2000 5221.5 (347.5)
>0.3000 5281.7 (383.8)
>0.4000 5434.1 (364.6)
>0.5000 5537.0 (471.9)
>0.6000 5767.4 (478.5)

```



```
>0.8000 6132.7 (472.5)
>1.0000 6593.6 (408.9)
```

```
Text(0.5, 0, 'Learning rate')
```



#### 11.2.4 Regularization (reg\_lambda) vs cross validation error

The parameter `reg_lambda` penalizes the  $L2$  norm of the leaf scores. For example, in case of classification, it will penalize the summation of the square of log odds of the predicted

probability. This penalization will tend to reduce the log odds, thereby reducing the tendency to overfit. “*Reducing the log odds*” in layman terms will mean not being overly sure about the prediction.

Without regularization, the algorithm will be closer to the gradient boosting algorithm. Regularization may provide some additional boost to prediction accuracy by reducing over-fitting. In the example below, regularization with *reg\_lambda=1 turns out to be better than no regularization* (reg\_lambda=0)\*. Of course, too much regularization may increase bias so much such that it leads to a decrease in prediction accuracy.

```
def get_models():
    models = dict()
    # explore 'reg_lambda' from 0.1 to 2 in 0.1 increments
    for i in [0,0.5,1.0,1.5,2,10,100]:
        key = '%.4f' % i
        models[key] = xgb.XGBRegressor(reg_lambda=i,random_state=1)
    return models

# evaluate a given model using cross-validation
def evaluate_model(model, X, y):
    # define the evaluation procedure
    cv = KFold(n_splits=10, shuffle=True, random_state=1)
    # evaluate the model and collect the results
    scores = np.sqrt(-cross_val_score(model, X, y, scoring='neg_mean_squared_error', cv=cv, n_jobs=-1))
    return scores

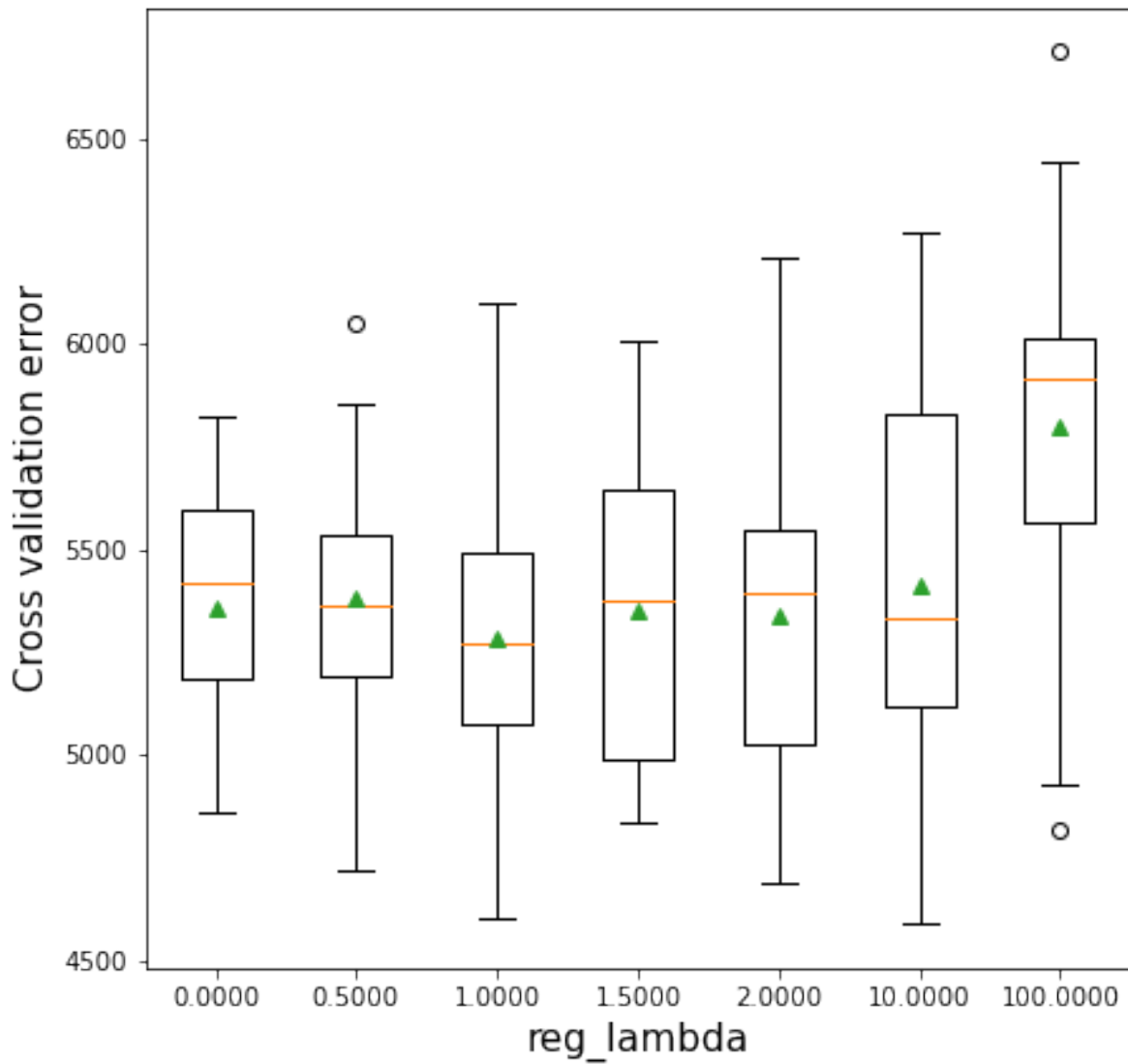
# get the models to evaluate
models = get_models()
# evaluate the models and store results
results, names = list(), list()
for name, model in models.items():
    # evaluate the model
    scores = evaluate_model(model, X, y)
    # store the results
    results.append(scores)
    names.append(name)
    # summarize the performance along the way
    print('>%s %.1f (%.1f)' % (name, np.mean(scores), np.std(scores)))
# plot model performance for comparison
plt.figure(figsize=(7, 7))
plt.boxplot(results, labels=names, showmeans=True)
plt.ylabel('Cross validation error',fontsize=15)
plt.xlabel('reg_lambda',fontsize=15)
```

```

>0.0000 5359.2 (317.0)
>0.5000 5382.7 (363.1)
>1.0000 5281.7 (383.8)
>1.5000 5348.0 (383.9)
>2.0000 5336.4 (426.6)
>10.0000 5410.9 (521.9)
>100.0000 5801.1 (563.7)

```

```
Text(0.5, 0, 'reg_lambda')
```



### 11.2.5 Regularization (gamma) vs cross validation error

The parameter `gamma` penalizes the tree based on the number of leaves. This is similar to the parameter `alpha` of cost complexity pruning. As `gamma` increases, more leaves will be pruned. Note that the previous parameter `reg_lambda` penalizes the leaf score, but does not prune the tree.

Without regularization, the algorithm will be closer to the gradient boosting algorithm. Regularization may provide some additional boost to prediction accuracy by reducing over-fitting. However, in the example below, no regularization (in terms of `gamma=0`) turns out to be better than a non-zero regularization. (*reg\_lambda=0*).

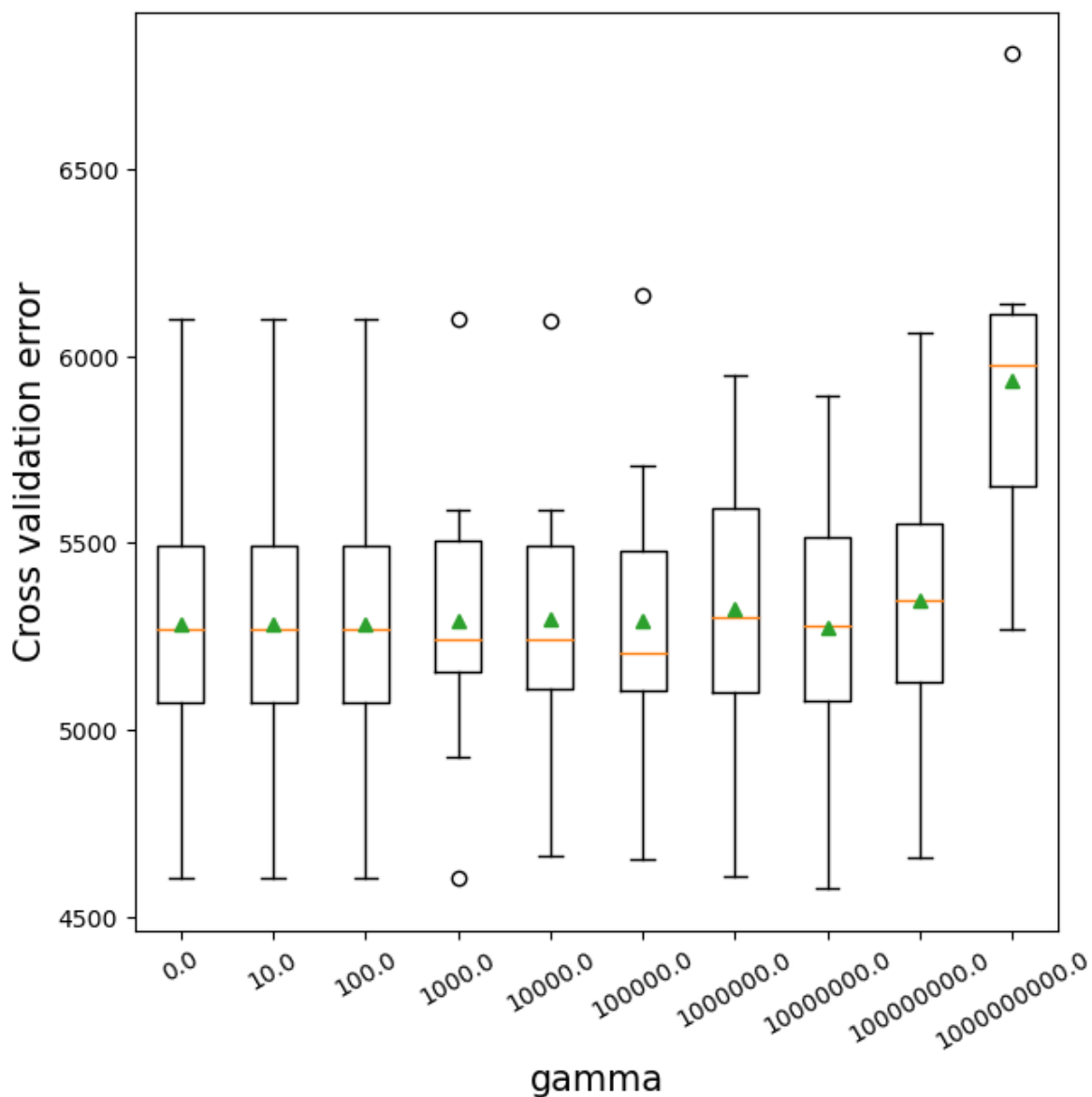
```
def get_models():
    models = dict()
    # explore gamma from 0.1 to 2 in 0.1 increments
    for i in [0,10,1e2,1e3,1e4,1e5,1e6,1e7,1e8,1e9]:
        key = '%.4f' % i
        models[key] = xgb.XGBRegressor(gamma=i,random_state=1)
    return models

# evaluate a given model using cross-validation
def evaluate_model(model, X, y):
    # define the evaluation procedure
    cv = KFold(n_splits=10, shuffle=True, random_state=1)
    # evaluate the model and collect the results
    scores = np.sqrt(-cross_val_score(model, X, y, scoring='neg_mean_squared_error', cv=cv, n_jobs=-1))
    return scores

# get the models to evaluate
models = get_models()
# evaluate the models and store results
results, names = list(), list()
for name, model in models.items():
    # evaluate the model
    scores = evaluate_model(model, X, y)
    # store the results
    results.append(scores)
    names.append(name)
    # summarize the performance along the way
    print('>%s %.1f (%.1f)' % (name, np.mean(scores), np.std(scores)))
# plot model performance for comparison
plt.figure(figsize=(7, 7))
plt.boxplot(results, labels=names, showmeans=True)
```

```
plt.ylabel('Cross validation error',fontsize=15)
plt.xlabel('gamma',fontsize=15)
#ax.set_xticklabels(x.astype(int))
plt.xticks(ticks=plt.xticks()[0].astype(int), labels=np.round([0,10,1e2,1e3,1e4,1e5,1e6,1e7,
rotation = 30);
```

```
>0.0000 5281.7 (383.8)
>10.0000 5281.7 (383.8)
>100.0000 5281.7 (383.8)
>1000.0000 5291.8 (381.8)
>10000.0000 5295.7 (370.2)
>100000.0000 5293.0 (402.5)
>1000000.0000 5322.2 (368.9)
>10000000.0000 5273.7 (409.8)
>100000000.0000 5345.4 (373.9)
>1000000000.0000 5932.3 (397.6)
```



### 11.2.6 Tuning XGboost regressor

Along with `max_depth`, `learning_rate`, and `n_estimators`, here we tune `reg_lambda` - the regularization parameter for penalizing the tree predictions.

```
#K-fold cross validation to find optimal parameters for XGBoost
start_time = time.time()
param_grid = {'max_depth': [4,6,8],
```

```

        'learning_rate': [0.01, 0.05, 0.1],
        'reg_lambda': [0, 1, 10],
        'n_estimators': [100, 500, 1000],
        'gamma': [0, 10, 100],
        'subsample': [0.5, 0.75, 1.0],
        'colsample_bytree': [0.5, 0.75, 1.0]}

cv = KFold(n_splits=5, shuffle=True, random_state=1)
optimal_params = RandomizedSearchCV(estimator=xgb.XGBRegressor(random_state=1),
                                    param_distributions = param_grid, n_iter = 200,
                                    verbose = 1,
                                    n_jobs=-1,
                                    cv = cv)

optimal_params.fit(X,y)
print("Optimal parameter values =", optimal_params.best_params_)
print("Optimal cross validation R-squared = ", optimal_params.best_score_)
print("Time taken = ", round((time.time()-start_time)/60), " minutes")

```

Fitting 5 folds for each of 200 candidates, totalling 1000 fits

Optimal parameter values = {'subsample': 0.75, 'reg\_lambda': 1, 'n\_estimators': 1000, 'max\_d

Optimal cross validation R-squared = 0.9002580404500382

Time taken = 4 minutes

#RMSE based on the optimal parameter values

```
np.sqrt(mean_squared_error(optimal_params.best_estimator_.predict(Xtest), ytest))
```

5497.553788113875

Let us use Bayes search to tune the model.

```

model = xgb.XGBRegressor(random_state = 1)

grid = {'max_leaves': Integer(4, 5000),
        'learning_rate': Real(0.0001, 1.0),
        'reg_lambda': Real(0, 1e4),
        'n_estimators': Integer(2, 2000),
        'gamma': Real(0, 1e11),
        'subsample': Real(0.1, 1.0),
        'colsample_bytree': Real(0.1, 1.0)}

```

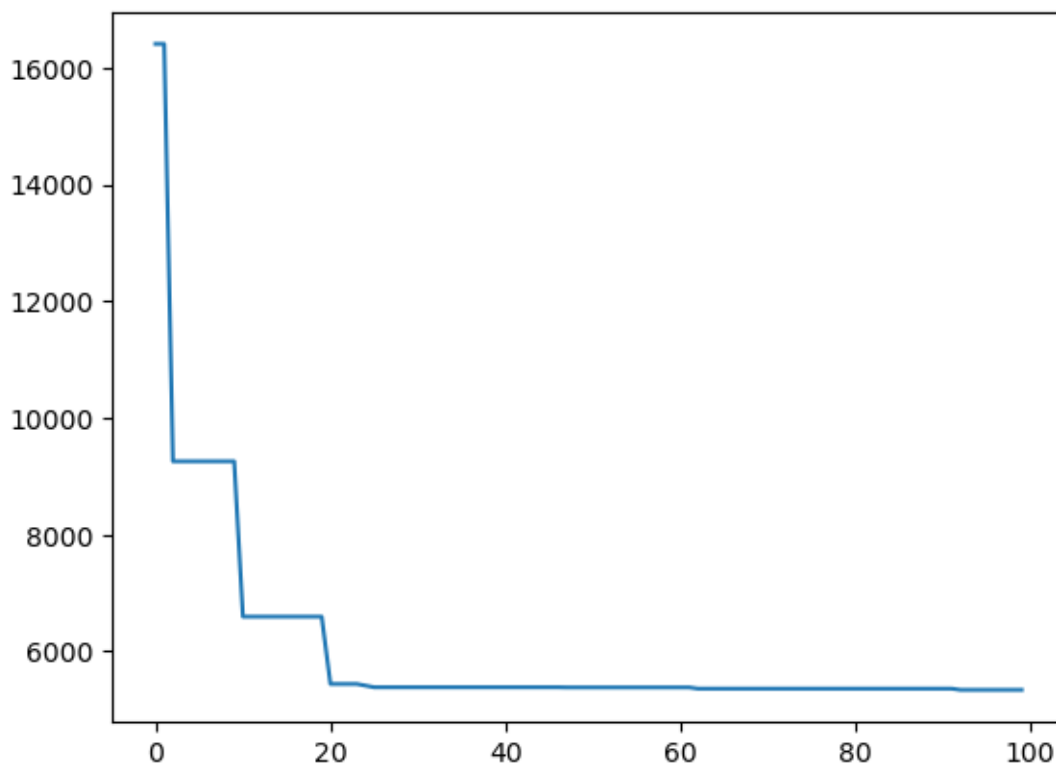
```

kfold = KFold(n_splits = 5, shuffle = True, random_state = 1)
gcv = BayesSearchCV(model, search_spaces = grid, cv = kfold, n_iter = 100, random_state = 1,
                    scoring = 'neg_root_mean_squared_error', n_jobs = -1)
paras = list(gcv.search_spaces.keys())
paras.sort()

def monitor(optim_result):
    cv_values = pd.Series(optim_result['func_vals']).cummin()
    display.clear_output(wait = True)
    min_ind = pd.Series(optim_result['func_vals']).argmin()
    print(paras, "=", optim_result['x_iters'][min_ind], pd.Series(optim_result['func_vals']))
    sns.lineplot(cv_values)
    plt.show()
gcv.fit(X, y, callback = monitor)

```

['colsample\_bytree', 'gamma', 'learning\_rate', 'max\_leaves', 'n\_estimators', 'reg\_lambda', '...



BayesSearchCV(cv=KFold(n\_splits=5, random\_state=1, shuffle=True),



```

estimator=XGBRegressor(base_score=None, booster=None,
                        callbacks=None, colsample_bylevel=None,
                        colsample_bynode=None,
                        colsample_bytree=None,
                        early_stopping_rounds=None,
                        enable_categorical=False, eval_metric=None,
                        feature_types=None, gamma=None,
                        gpu_id=None, grow_policy=None,
                        importance_type=None,
                        inte...
                        'learning_rate': Real(low=0.0001, high=1.0, prior='uniform', tra
                        'max_leaves': Integer(low=4, high=5000, prior='uniform', trans
                        'n_estimators': Integer(low=2, high=2000, prior='uniform', tran
                        'reg_lambda': Real(low=0, high=10000.0, prior='uniform', trans
                        'subsample': Real(low=0.1, high=1.0, prior='uniform', transform

```

```

model1 = xgb.XGBRegressor(random_state = 1, colsample_bytree = 0.85, gamma = 0, learning_rate = 0.1,
                           max_leaves = 802, n_estimators = 1023, reg_lambda = 1394, subsample = 0.8)

```

```

np.sqrt(mean_squared_error(model1.predict(Xtest),ytest))

```

5466.076861800755

We got a different set of optimal hyperparameters with Bayes search. Thus, ensembling the model based on the two sets of hyperparameters is likely to improve the accuracy over the individual models.

```

model2 = xgb.XGBRegressor(random_state = 1, colsample_bytree = 1.0, gamma = 100, learning_rate = 0.1,
                           max_depth = 8, n_estimators = 1000, reg_lambda = 1, subsample = 0.8)

```

```

np.sqrt(mean_squared_error(0.5*model1.predict(Xtest)+0.5*model2.predict(Xtest),ytest))

```

5393.379834226845

### 11.2.7 Early stopping with XGBoost

If we have a test dataset (*or we can further split the train data into a smaller train and test data*), we can use it with the `early_stopping_rounds` argument of XGBoost, where it will stop growing trees once the model accuracy fails to increase for a certain number of consecutive iterations, given as `early_stopping_rounds`.

```
X_train_sub, X_test_sub, y_train_sub, y_test_sub = \
train_test_split(X, y, test_size = 0.2, random_state = 45)
```

```
model = xgb.XGBRegressor(random_state = 1, max_depth = 8, learning_rate = 0.01,
                        n_estimators = 20000, reg_lambda = 1, gamma = 100, subsample = 0.75,
model.fit(X_train_sub, y_train_sub, eval_set = ((X_test_sub, y_test_sub))), early_stopping_
```

The results of the code are truncated to save space. A snapshot of the beginning and end of the results is below. The algorithm keeps adding trees to the model until the RMSE ceases to decrease for 250 consecutive iterations.

```
<IPython.core.display.Image object>
```

```
print("XGBoost RMSE = ", np.sqrt(mean_squared_error(model.predict(Xtest), ytest)))
```

```
XGBoost RMSE = 5508.787454011525
```

Let us further reduce the learning rate to 0.001 and see if the accuracy increases further on the test data. We'll use the `early_stopping_rounds` argument to stop growing trees once the accuracy fails to increase for 250 consecutive iterations.

```
model = xgb.XGBRegressor(random_state = 1, max_depth = 8, learning_rate = 0.001,
                        n_estimators = 20000, reg_lambda = 1, gamma = 100, subsample = 0.75,
model.fit(X_train_sub, y_train_sub, eval_set = ((X_test_sub, y_test_sub))), early_stopping_
```

```
<IPython.core.display.Image object>
```

```
print("XGBoost RMSE = ", np.sqrt(mean_squared_error(model.predict(Xtest), ytest)))
```

```
XGBoost RMSE = 5483.518711988693
```

Note that the accuracy on this test data has further increased with a lower learning rate.

Let us combine the XGBoost model with other tuned models from earlier chapters.

```

#Tuned AdaBoost model from Section 7.2.4
model_ada = AdaBoostRegressor(base_estimator=DecisionTreeRegressor(max_depth=10),n_estimators=100,
                             random_state=1).fit(X,y)
print("AdaBoost RMSE = ", np.sqrt(mean_squared_error(model_ada.predict(Xtest),ytest)))

#Tuned Random forest model from Section 6.1.2
model_rf = RandomForestRegressor(n_estimators=300, random_state=1,
                                n_jobs=-1, max_features=2).fit(X, y)
print("Random Forest RMSE = ",np.sqrt(mean_squared_error(model_rf.predict(Xtest),ytest)))

#Tuned gradient boosting model from Section 8.2.5
model_gb = GradientBoostingRegressor(max_depth=8,n_estimators=100,learning_rate=0.1,
                                     random_state=1,loss='huber').fit(X,y)
print("Gradient boost RMSE = ",np.sqrt(mean_squared_error(model_gb.predict(Xtest),ytest)))

```

```

AdaBoost RMSE = 5693.165811600585
Random Forest RMSE = 5642.45839697972
Gradient boost RMSE = 5405.787029062213

```

```

#Ensemble model
pred_xgb = model.predict(Xtest)    #XGBoost
pred_ada = model_ada.predict(Xtest)#AdaBoost
pred_rf = model_rf.predict(Xtest)  #Random Forest
pred_gb = model_gb.predict(Xtest)  #Gradient boost
pred = 0.25*pred_xgb + 0.25*pred_ada + 0.25*pred_rf + 0.25*pred_gb #Option 1 - All models are given equal weight
#pred = 0.15*pred1+0.15*pred2+0.15*pred3+0.55*pred4 #Option 2 - Higher weight to the better model
print("Ensemble model RMSE = ", np.sqrt(mean_squared_error(pred,ytest)))

```

```

Ensemble model RMSE = 5352.145010078119

```

Combined, the random forest model, gradient boost, XGBoost and the Adaboost model do better than each of the individual models.

## 11.3 XGBoost for classification

```

data = pd.read_csv('./Datasets/Heart.csv')
data.dropna(inplace = True)
data.head()

```

	Age	Sex	ChestPain	RestBP	Chol	Fbs	RestECG	MaxHR	ExAng	Oldpeak	Slope	Ca
0	63	1	typical	145	233	1	2	150	0	2.3	3	0.0
1	67	1	asymptomatic	160	286	0	2	108	1	1.5	2	3.0
2	67	1	asymptomatic	120	229	0	2	129	1	2.6	2	2.0
3	37	1	nonanginal	130	250	0	0	187	0	3.5	3	0.0
4	41	0	nontypical	130	204	0	2	172	0	1.4	1	0.0

```
#Response variable
y = pd.get_dummies(data['AHD'])['Yes']

#Creating a dataframe for predictors with dummy variables replacing the categorical variables
X = data.drop(columns = ['AHD','ChestPain','Thal'])
X = pd.concat([X,pd.get_dummies(data['ChestPain']),pd.get_dummies(data['Thal'])],axis=1)
X.head()
```

	Age	Sex	RestBP	Chol	Fbs	RestECG	MaxHR	ExAng	Oldpeak	Slope	Ca	asymptomatic
0	63	1	145	233	1	2	150	0	2.3	3	0.0	0
1	67	1	160	286	0	2	108	1	1.5	2	3.0	1
2	67	1	120	229	0	2	129	1	2.6	2	2.0	1
3	37	1	130	250	0	0	187	0	3.5	3	0.0	0
4	41	0	130	204	0	2	172	0	1.4	1	0.0	0

```
#Creating train and test datasets
Xtrain,Xtest,ytrain,ytest = train_test_split(X,y,train_size = 0.5,random_state=1)
```

XGBoost has an additional parameter for classification: **scale\_pos\_weight**

Gradients are used as the basis for fitting subsequent trees added to boost or correct errors made by the existing state of the ensemble of decision trees.

The **scale\_pos\_weight** value is used to scale the gradient for the positive class.

This has the effect of scaling errors made by the model during training on the positive class and encourages the model to over-correct them. In turn, this can help the model achieve better performance when making predictions on the positive class. Pushed too far, it may result in the model overfitting the positive class at the cost of worse performance on the negative class or both classes.

As such, the **scale\_pos\_weight** hyperparameter can be used to train a class-weighted or cost-sensitive version of XGBoost for imbalanced classification.

A sensible default value to set for the `scale_pos_weight` hyperparameter is the inverse of the class distribution. For example, for a dataset with a 1 to 100 ratio for examples in the minority to majority classes, the `scale_pos_weight` can be set to 100. This will give classification errors made by the model on the minority class (positive class) 100 times more impact, and in turn, 100 times more correction than errors made on the majority class.

## Reference

```
start_time = time.time()
param_grid = {'n_estimators': [25, 100, 500],
              'max_depth': [6, 7, 8],
              'learning_rate': [0.01, 0.1, 0.2],
              'gamma': [0.1, 0.25, 0.5],
              'reg_lambda': [0, 0.01, 0.001],
              'scale_pos_weight': [1.25, 1.5, 1.75] #Control the balance of positive and negative
            }

cv = StratifiedKFold(n_splits=5, shuffle=True, random_state=1)
optimal_params = GridSearchCV(estimator=xgb.XGBClassifier(objective = 'binary:logistic', random_state=1,
                                                         use_label_encoder=False),
                             param_grid = param_grid,
                             scoring = 'accuracy',
                             verbose = 1,
                             n_jobs=-1,
                             cv = cv)

optimal_params.fit(Xtrain, ytrain)
print(optimal_params.best_params_, optimal_params.best_score_)
print("Time taken = ", (time.time()-start_time)/60, " minutes")
```

Fitting 5 folds for each of 729 candidates, totalling 3645 fits

[22:00:02] WARNING: D:\bld\xgboost-split\_1645118015404\work\src\learner.cc:1115: Starting in {'gamma': 0.25, 'learning\_rate': 0.2, 'max\_depth': 6, 'n\_estimators': 25, 'reg\_lambda': 0.01

```
cv_results=pd.DataFrame(optimal_params.cv_results_)
cv_results.sort_values(by = 'mean_test_score', ascending=False)[0:5]
```

	mean_fit_time	std_fit_time	mean_score_time	std_score_time	param_gamma	param_learning_rate
409	0.111135	0.017064	0.005629	0.000737	0.25	0.2
226	0.215781	0.007873	0.005534	0.001615	0.1	0.2
290	1.391273	0.107808	0.007723	0.006286	0.25	0.01
266	1.247463	0.053597	0.006830	0.002728	0.25	0.01

	mean_fit_time	std_fit_time	mean_score_time	std_score_time	param_gamma	param_learning_rate
269	1.394361	0.087307	0.005530	0.001718	0.25	0.01

```
#Function to compute confusion matrix and prediction accuracy on test/train data
def confusion_matrix_data(data,actual_values,model,cutoff=0.5):
#Predict the values using the Logit model
    pred_values = model.predict_proba(data)[:,-1]
# Specify the bins
    bins=np.array([0,cutoff,1])
#Confusion matrix
    cm = np.histogram2d(actual_values, pred_values, bins=bins)[0]
    cm_df = pd.DataFrame(cm)
    cm_df.columns = ['Predicted 0','Predicted 1']
    cm_df = cm_df.rename(index={0: 'Actual 0',1:'Actual 1'})
# Calculate the accuracy
    accuracy = 100*(cm[0,0]+cm[1,1])/cm.sum()
    fnr = 100*(cm[1,0])/(cm[1,0]+cm[1,1])
    precision = 100*(cm[1,1])/(cm[0,1]+cm[1,1])
    fpr = 100*(cm[0,1])/(cm[0,0]+cm[0,1])
    tpr = 100*(cm[1,1])/(cm[1,0]+cm[1,1])
    print("Accuracy = ", accuracy)
    print("Precision = ", precision)
    print("FNR = ", fnr)
    print("FPR = ", fpr)
    print("TPR or Recall = ", tpr)
    print("Confusion matrix = \n", cm_df)
    return (" ")
```

```
model4 = xgb.XGBClassifier(objective = 'binary:logistic',random_state=1,gamma=0.25,learning_rate=0.01,
                           n_estimators = 500,reg_lambda = 0.01,scale_pos_weight=1.75)
model4.fit(Xtrain,ytrain)
model4.score(Xtest,ytest)
```

0.7718120805369127

```
#Computing the accuracy
y_pred = model4.predict(Xtest)
print("Accuracy: ",accuracy_score(y_pred, ytest)*100)

#Computing the ROC-AUC
```

```

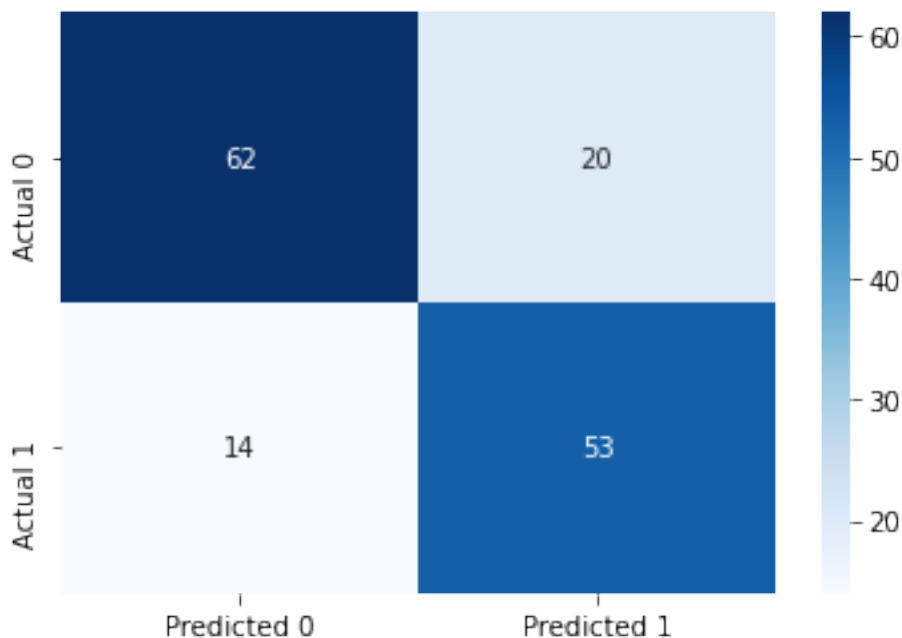
y_pred_prob = model4.predict_proba(Xtest)[: ,1]
fpr, tpr, auc_thresholds = roc_curve(ytest, y_pred_prob)
print("ROC-AUC: ",auc(fpr, tpr))# AUC of ROC

#Computing the precision and recall
print("Precision: ", precision_score(ytest, y_pred))
print("Recall: ", recall_score(ytest, y_pred))

#Confusion matrix
cm = pd.DataFrame(confusion_matrix(ytest, y_pred), columns=['Predicted 0', 'Predicted 1'],
                  index = ['Actual 0', 'Actual 1'])
sns.heatmap(cm, annot=True, cmap='Blues', fmt='g');

```

Accuracy: 77.18120805369128  
 ROC-AUC: 0.8815070986530761  
 Precision: 0.726027397260274  
 Recall: 0.7910447761194029



If we increase the value of `scale_pos_weight`, the model will focus on classifying positives more correctly. This will increase the recall (true positive rate) since the focus is on identifying all positives. However, this will lead to identifying positives aggressively, and observations 'similar' to observations of the positive class will also be predicted as positive resulting in an

increase in false positives and a decrease in precision. See the trend below as we increase the value of `scale_pos_weight`.

### 11.3.1 Precision & recall vs `scale_pos_weight`

```
def get_models():
    models = dict()
    # explore 'scale_pos_weight' from 0.1 to 2 in 0.1 increments
    for i in [0,1,10,1e2,1e3,1e4,1e5,1e6,1e7,1e8,1e9]:
        key = '%.0f' % i
        models[key] = xgb.XGBClassifier(objective = 'binary:logistic',scale_pos_weight=i,random_state=1)
    return models

# evaluate a given model using cross-validation
def evaluate_model(model, X, y):
    # define the evaluation procedure
    cv = StratifiedKFold(n_splits=5, shuffle=True, random_state=1)
    # evaluate the model and collect the results
    scores_recall = cross_val_score(model, X, y, scoring='recall', cv=cv, n_jobs=-1)
    scores_precision = cross_val_score(model, X, y, scoring='precision', cv=cv, n_jobs=-1)
    return list([scores_recall,scores_precision])

# get the models to evaluate
models = get_models()
# evaluate the models and store results
results_recall, results_precision, names = list(), list(), list()
for name, model in models.items():
    # evaluate the model
    scores = evaluate_model(model, X, y)
    scores_recall = scores[0]
    scores_precision = scores[1]
    # store the results
    results_recall.append(scores_recall)
    results_precision.append(scores_precision)
    names.append(name)
    # summarize the performance along the way
    print('>%s %.2f (%.2f)' % (name, np.mean(scores_recall), np.std(scores_recall)))
# plot model performance for comparison
plt.figure(figsize=(7, 7))
sns.set(font_scale = 1.5)
pdata = pd.DataFrame(results_precision)
```



```

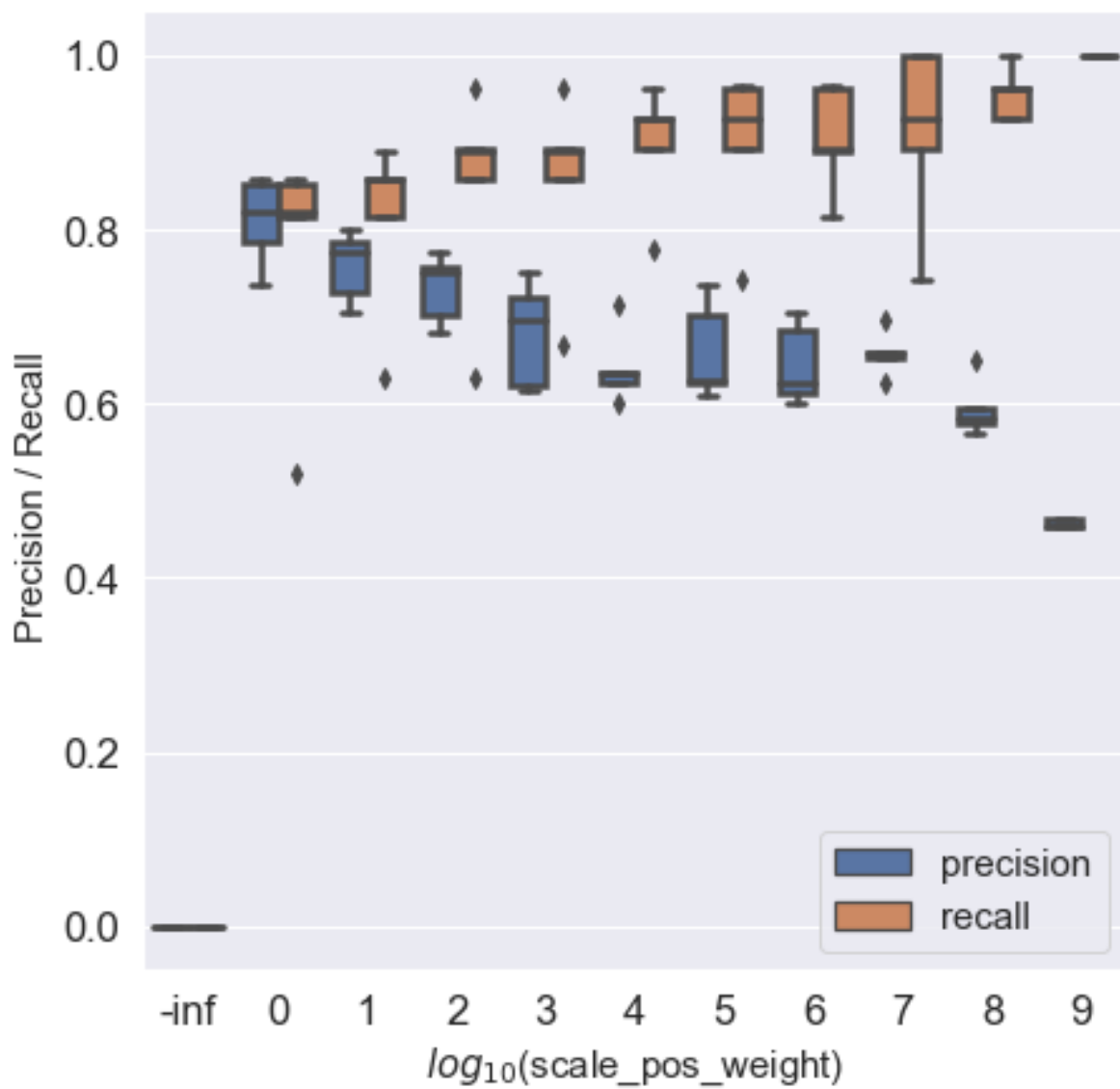
pdata.columns = list(['p1','p2','p3','p4','p5'])
pdata['metric'] = 'precision'
rdata = pd.DataFrame(results_recall)
rdata.columns = list(['p1','p2','p3','p4','p5'])
rdata['metric'] = 'recall'
pr_data = pd.concat([pdata,rdata])
pr_data.reset_index(drop=False,inplace= True)
#sns.boxplot(x="day", y="total_bill", hue="time",pr_data=tips, linewidth=2.5)
pr_data_melt=pr_data.melt(id_vars = ['index','metric'])
pr_data_melt['index']=pr_data_melt['index']-1
pr_data_melt['index'] = pr_data_melt['index'].astype('str')
pr_data_melt.replace(to_replace='-1',value =  '-inf',inplace=True)
sns.boxplot(x='index', y="value", hue="metric", data=pr_data_melt, linewidth=2.5)
plt.xlabel('$log_{10}$(scale_pos_weight)',fontsize=15)
plt.ylabel('Precision / Recall ',fontsize=15)
plt.legend(loc="lower right", frameon=True, fontsize=15)

```

```

>0 0.00 (0.00)
>1 0.77 (0.13)
>10 0.81 (0.09)
>100 0.85 (0.11)
>1000 0.85 (0.10)
>10000 0.90 (0.06)
>100000 0.90 (0.08)
>1000000 0.90 (0.06)
>10000000 0.91 (0.10)
>100000000 0.96 (0.03)
>1000000000 1.00 (0.00)

```



## 12 LightGBM and CatBoost

```
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import cross_val_score, train_test_split, KFold, cross_val_predict
from sklearn.metrics import mean_squared_error, r2_score, roc_curve, auc, precision_recall_curve
recall_score, precision_score, confusion_matrix
from sklearn.tree import DecisionTreeRegressor, DecisionTreeClassifier
from sklearn.model_selection import GridSearchCV, ParameterGrid, StratifiedKFold, RandomizedSearchCV
from sklearn.ensemble import VotingRegressor, VotingClassifier, StackingRegressor, StackingClassifier
from sklearn.linear_model import LinearRegression, LogisticRegression, LassoCV, RidgeCV, ElasticNetCV
from sklearn.neighbors import KNeighborsRegressor
import itertools as it
import time as time
import xgboost as xgb
from lightgbm import LGBMRegressor
from catboost import CatBoostRegressor
from skopt import BayesSearchCV
from skopt.space import Real, Categorical, Integer
from skopt.plots import plot_objective, plot_histogram, plot_convergence
import warnings
from IPython import display
```

We'll continue to use the same datasets that we have been using throughout the course.

```
#Using the same datasets as used for linear regression in STAT303-2,
#so that we can compare the non-linear models with linear regression
trainf = pd.read_csv('./Datasets/Car_features_train.csv')
trainp = pd.read_csv('./Datasets/Car_prices_train.csv')
testf = pd.read_csv('./Datasets/Car_features_test.csv')
testp = pd.read_csv('./Datasets/Car_prices_test.csv')
train = pd.merge(trainf, trainp)
```

```
test = pd.merge(testf, testp)
train.head()
```

	carID	brand	model	year	transmission	mileage	fuelType	tax	mpg	engineSize	price
0	18473	bmw	6 Series	2020	Semi-Auto	11	Diesel	145	53.3282	3.0	37980
1	15064	bmw	6 Series	2019	Semi-Auto	10813	Diesel	145	53.0430	3.0	33980
2	18268	bmw	6 Series	2020	Semi-Auto	6	Diesel	145	53.4379	3.0	36850
3	18480	bmw	6 Series	2017	Semi-Auto	18895	Diesel	145	51.5140	3.0	25998
4	18492	bmw	6 Series	2015	Automatic	62953	Diesel	160	51.4903	3.0	18990

```
X = train[['mileage', 'mpg', 'year', 'engineSize']]
Xtest = test[['mileage', 'mpg', 'year', 'engineSize']]
y = train['price']
ytest = test['price']
```

## 12.1 LightGBM

LightGBM is a gradient boosting decision tree algorithm developed by Microsoft in 2017. LightGBM outperforms XGBoost in terms of computational speed, and provides comparable accuracy in general. The following two key features in LightGBM that make it faster than XGBoost:

**1. Gradient-based One-Side Sampling (GOSS):** Recall, in gradient boosting, we fit trees on the gradient of the loss function (*refer the gradient boosting algorithm in section 10.10.2 of the book, [Elements of Statistical Learning](#)*):

$$r_m = - \left[ \frac{\partial L(y_i, f(x_i))}{\partial f(x_i)} \right]_{f=f_{m-1}}.$$

Observations that correspond to relatively larger gradients contribute more to minimizing the loss function as compared to observations with smaller gradients. The algorithm down-samples the observations with small gradients, while selecting all the observations with large gradients. As observations with large gradients contribute the most to the reduction in loss function when considering a split, the accuracy of loss reduction estimate is maintained even with a reduced sample size. This leads to similar performance in terms of prediction accuracy while reducing computation speed due to reduction in sample size to fit trees.

**2. Exclusive feature bundling (EFB):** This is useful when there are a lot of predictors, but the predictor space is sparse, i.e., most of the values are zero for several predictors, and

the predictors rarely take non-zero values simultaneously. This can typically happen in case of a lot of dummy variables in the data. In such a case, the predictors are bundled to create a single predictor.

In the example below you can see that feature1 and feature2 are mutually exclusive. In order to achieve non overlapping buckets we add bundle size of feature1 to feature2. This makes sure that non zero data points of bundled features (feature1 and feature2) reside in different buckets. In feature\_bundle buckets 1 to 4 contains non zero instances of feature1 and buckets 5,6 contain non zero instances of feature2 ([Reference](#)).

feature1	feature2	feature_bundle
0	2	6
0	1	5
0	2	6
1	0	1
2	0	2
3	0	3
4	0	4

Read the [LightGBM paper](#) for more details.

### 12.1.1 LightGBM for regression

Let us tune a lightGBM model for regression for our problem of predicting car price. We'll use the function [LGBMRegressor](#). For classification problems, [LGBMClassifier](#) can be used. Note that we are using the GOSS algorithm to downsample observations with smaller gradients.

```
#K-fold cross validation to find optimal parameters for LightGBM regressor
start_time = time.time()
param_grid = {'num_leaves': [20, 31, 40],
              'learning_rate': [0.01, 0.05, 0.1],
              'reg_lambda': [0, 10, 100],
              'n_estimators': [100, 500, 1000],
              'reg_alpha': [0, 10, 100],
              'subsample': [0.5, 0.75, 1.0],
              'colsample_bytree': [0.5, 0.75, 1.0]}

cv = KFold(n_splits=5, shuffle=True, random_state=1)
optimal_params = RandomizedSearchCV(estimator=LGBMRegressor(boosting_type = 'goss'),
                                   param_distributions = param_grid, n_iter = 200,
                                   verbose = 1, scoring='neg_root_mean_squared_error',
```

```

n_jobs=-1,random_state=1,
cv = cv)

optimal_params.fit(X,y)
print("Optimal parameter values =", optimal_params.best_params_)
print("Optimal cross validation RMSE = ",optimal_params.best_score_)
print("Time taken = ", round((time.time()-start_time)/60), " minutes")

```

Fitting 5 folds for each of 200 candidates, totalling 1000 fits

Optimal parameter values = {'subsample': 1.0, 'reg\_lambda': 10, 'reg\_alpha': 0, 'num\_leaves':

Optimal cross validation R-squared = -5670.309021679375

Time taken = 1 minutes

```

#RMSE based on the optimal parameter values of a LighGBM Regressor model
np.sqrt(mean_squared_error(optimal_params.best_estimator_.predict(Xtest),ytest))

```

5614.374498193448

Note that downsampling of small-gradient observations leads to faster execution time, but potentially by compromising some accuracy. We can expect to improve the accuracy by increasing the `top_rate` or the `other_rate` hyperparameters, but at an increased computational cost. In the cross-validation below, we have increased the `top_rate` to 0.5 from the default value of 0.2.

```

#K-fold cross validation to find optimal parameters for LightGBM regressor
start_time = time.time()
param_grid = {'num_leaves': [20, 31, 40],
              'learning_rate': [0.01, 0.05, 0.1],
              'reg_lambda': [0, 10, 100],
              'n_estimators': [100, 500, 1000],
              'reg_alpha': [0, 10, 100],
              'subsample': [0.5, 0.75, 1.0],
              'colsample_bytree': [0.5, 0.75, 1.0]}

cv = KFold(n_splits=5,shuffle=True,random_state=1)
optimal_params = RandomizedSearchCV(estimator=LGBMRegressor(boosting_type = 'goss', top_rate
                                param_distributions = param_grid, n_iter = 200,
                                verbose = 1, scoring='neg_root_mean_squared_error',
                                n_jobs=-1,random_state=1,
                                cv = cv)

optimal_params.fit(X,y)

```

```
print("Optimal parameter values =", optimal_params.best_params_)
print("Optimal cross validation RMSE = ", optimal_params.best_score_)
print("Time taken = ", round((time.time()-start_time)/60), " minutes")
```

Fitting 5 folds for each of 200 candidates, totalling 1000 fits

Optimal parameter values = {'subsample': 0.5, 'reg\_lambda': 0, 'reg\_alpha': 100, 'num\_leaves': 100}

Optimal cross validation R-squared = -5436.062435616846

Time taken = 1 minutes

#RMSE based on the optimal parameter values of a LighGBM Regressor model

```
np.sqrt(mean_squared_error(optimal_params.best_estimator_.predict(Xtest), ytest))
```

5355.964600884197

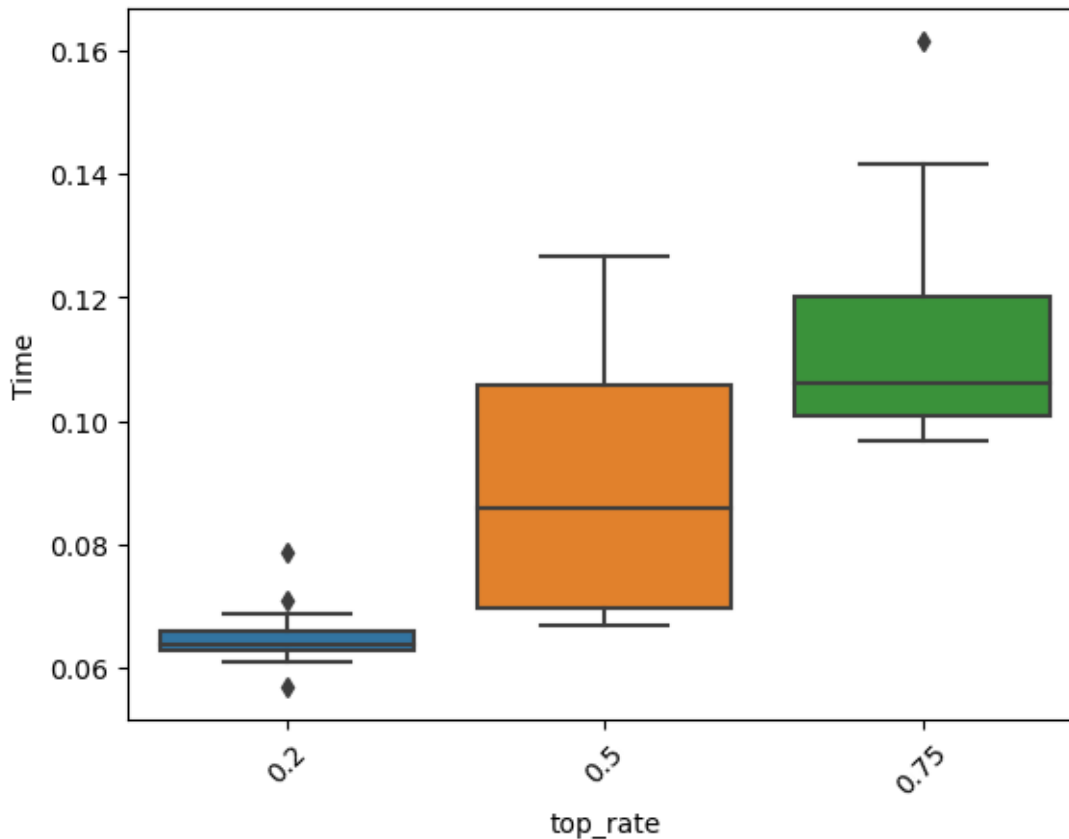
Note that the cross-validated RMSE has reduced. However, this is at an increased computational expense. In the simulations below, we compare the time taken to train models with increasing values of the `top_rate` hyperparameter.

```
time_list = []
for i in range(50):
    start_time = time.time()
    model = LGBMRegressor(boosting_type = 'goss', top_rate = 0.2, n_jobs=-1).fit(X, y)
    time_list.append(time.time()-start_time)
```

```
time_list2 = []
for i in range(50):
    start_time = time.time()
    model = LGBMRegressor(boosting_type = 'goss', top_rate = 0.5, n_jobs=-1).fit(X, y)
    time_list2.append(time.time()-start_time)
```

```
time_list3 = []
for i in range(50):
    start_time = time.time()
    model = LGBMRegressor(boosting_type = 'goss', top_rate = 0.8, n_jobs=-1).fit(X, y)
    time_list3.append(time.time()-start_time)
```

```
ax = sns.boxplot([time_list, time_list2, time_list3]);
ax.set_xticklabels([0.2, 0.5, 0.75]);
plt.ylabel('Time');
plt.xlabel('top_rate');
plt.xticks(rotation = 45);
```



### 12.1.2 LightGBM vs XGBoost

LightGBM model took 2 minutes for a random search with 1000 fits as compared to 7 minutes for an XGBoost model with 1000 fits on the same data (as shown below). In terms of prediction accuracy, we observe that the accuracy of LightGBM on test (*unseen*) data is comparable to that of XGBoost.

```
#K-fold cross validation to find optimal parameters for XGBoost
start_time = time.time()
param_grid = {'max_depth': [4,6,8],
              'learning_rate': [0.01, 0.05, 0.1],
              'reg_lambda': [0, 1, 10],
              'n_estimators': [100, 500, 1000],
              'gamma': [0, 10, 100],
              'subsample': [0.5, 0.75, 1.0],
              'colsample_bytree': [0.5, 0.75, 1.0]}
```



```

cv = KFold(n_splits=5,shuffle=True,random_state=1)
optimal_params = RandomizedSearchCV(estimator=xgb.XGBRegressor(),
                                   param_distributions = param_grid, n_iter = 200,
                                   verbose = 1, scoring = 'neg_root_mean_squared_error',
                                   n_jobs=-1,random_state = 1,
                                   cv = cv)

optimal_params.fit(X,y)
print("Optimal parameter values =", optimal_params.best_params_)
print("Optimal cross validation R-squared = ",optimal_params.best_score_)
print("Time taken = ", round((time.time()-start_time)/60), " minutes")

```

Fitting 5 folds for each of 200 candidates, totalling 1000 fits

Optimal parameter values = {'subsample': 0.75, 'reg\_lambda': 1, 'n\_estimators': 1000, 'max\_d

Optimal cross validation R-squared = -5178.8689594137295

Time taken = 7 minutes

#RMSE based on the optimal parameter values

```
np.sqrt(mean_squared_error(optimal_params.best_estimator_.predict(Xtest),ytest))
```

5420.661056398766

## 12.2 CatBoost

CatBoost is a gradient boosting algorithm developed by Yandex (*Russian Google*) in 2017. Like LightGBM, CatBoost is also faster than XGBoost in training. However, unlike LightGBM, the authors have claimed that it outperforms both LightGBM and XGBoost in terms of prediction accuracy as well.

The key feature of CatBoost that address the issue with the gradient boosting procedure is the idea of ordered boosting. Classic boosting algorithms are prone to overfitting on small/noisy datasets due to a problem known as prediction shift. Recall, in gradient boosting, we fit trees on the gradient of the loss function (*refer the gradient boosting algorithm in section 10.10.2 of the book, [Elements of Statistical Learning](#)*):

$$r_m = - \left[ \frac{\partial L(y_i, f(x_i))}{\partial f(x_i)} \right]_{f=f_{m-1}}.$$

When calculating the gradient estimate of an observation, these algorithms use the same observations that the model was built with, thus having no chances of experiencing unseen

data. CatBoost, on the other hand, uses the concept of ordered boosting, a permutation-driven approach to train model on a subset of data while calculating residuals on another subset, thus preventing “target leakage” and overfitting. The residuals of an observation are computed based on a model developed on the previous observations, where the observations are randomly shuffled at each iteration, i.e., for each tree.

Thus, the gradient of the loss function is based on test (*unseen*) data, instead of the data on which the model has been trained, which improves the generalizability of the model, and avoids overfitting on train data.

The authors have also shown that CatBoost performs better than XGBoost and LightGBM without tuning, i.e., with default hyperparameter settings.

Read the [CatBoost paper](#) for more details.

Here is a good [blog](#) listing the key features of CatBoost.

### 12.2.1 CatBoost for regression

We’ll use the function [CatBoostRegressor](#) for regression. For classification problems [CatBoostClassifier](#) can be used.

Let us check the performance of `CatBoostRegressor()` without tuning, i.e., with default hyperparameter settings.

```
model_cat = CatBoostRegressor(verbose=0).fit(X, y)

cv = KFold(n_splits=5, shuffle=True, random_state=1)
np.mean(-cross_val_score(CatBoostRegressor(verbose=0), X, y, cv = cv, n_jobs = -1,
                          scoring='neg_root_mean_squared_error'))
```

```
5035.972129299527
```

```
np.sqrt(mean_squared_error(model_cat.predict(Xtest), ytest))
```

```
5288.82153844634
```

Even with default hyperparameter settings, CatBoost has outperformed both XGBoost and LightGBM in terms of cross-validated RMSE, and RMSE on test data for our example of predicting car prices.

## 12.2.2 Target encoding with CatBoost

Target encoding for categorical variables can be used with CatBoost, that may benefit in terms of both speed and accuracy. However, the benefit is not guaranteed. Let us use target encoding for the categorical predictors `brand`, `model`, `transmission` and `fuelType`.

```
X = train[['mileage','mpg','year','engineSize', 'brand', 'model', 'transmission', 'fuelType']]
Xtest = test[['mileage','mpg','year','engineSize', 'brand', 'model', 'transmission', 'fuelType']]
y = train['price']
ytest = test['price']
```

The parameter `cat_features` will be used to specify the indices of the categorical predictors for target encoding.

```
model = CatBoostRegressor(verbose = False, cat_features = range(4, 8)).fit(X, y)
mean_squared_error(model.predict(Xtest), ytest, squared = False)
```

3263.1348853593345

Let us compare the results with one-hot encoding of the categorical predictors.

```
X = train[['mileage','mpg','year','engineSize', 'brand', 'model', 'transmission', 'fuelType']]
Xtest = test[['mileage','mpg','year','engineSize', 'brand', 'model', 'transmission', 'fuelType']]
X = pd.get_dummies(X)
Xtest = pd.get_dummies(Xtest)
```

In one-hot encoding, we need to make sure that both the datasets have the same predictors. Let us find the predictors in train data that are not in test data. Note that this is not necessary in target encoding.

```
np.setdiff1d(X.columns, Xtest.columns)
```

```
array(['model_ M6'], dtype=object)
```

```
X.drop(columns = 'model_ M6', inplace = True)
y = train['price']
ytest = test['price']
```

```
model = CatBoostRegressor(verbose = False).fit(X, y)
mean_squared_error(model.predict(Xtest), ytest, squared = False)
```

3219.857899121199

In this case, target encoding has a slightly higher RMSE as compared to one-hot encoding. However, it may do better than one-hot-encoding in a different problem.

Let us use both target encoding and one-hot encoding together to see if it helps do better than each of the them individually.

```
X = pd.concat([train[['brand', 'model', 'transmission', 'fuelType']], X], axis = 1)
Xtest = pd.concat([test[['brand', 'model', 'transmission', 'fuelType']], Xtest], axis = 1)
```

```
model = CatBoostRegressor(verbose = False, cat_features = range(4)).fit(X, y)
mean_squared_error(model.predict(Xtest), ytest, squared = False)
```

3172.449374536484

In this case, using target-encoding and one-hot-encoding together does better on test data. Using both the encodings together will help reduce bias while increasing variance. The benefit of using both the encodings together depends on the bias-variance tradeoff.

### 12.2.3 CatBoost vs XGBoost

Let us see the performance of XGBoost with default hyperparameter settings.

```
model_xgb = xgb.XGBRFRegressor().fit(X, y)
np.mean(-cross_val_score(xgb.XGBRFRegressor(), X, y, cv = cv, n_jobs = -1,
                        scoring='neg_root_mean_squared_error'))
```

6273.043859096154

```
np.sqrt(mean_squared_error(model_xgb.predict(Xtest), ytest))
```

6821.745153860935

XGBoost performance deteriorates showing that hyperparameter tuning is more important in XGBoost.

Let us see the performance of LightGBM with default hyperparameter settings.

```
model_lgbm = LGBMRegressor().fit(X, y)
np.mean(-cross_val_score(LGBMRegressor(), X, y, cv = cv, n_jobs = -1,
                          scoring='neg_root_mean_squared_error'))
```

5562.149251902867

```
np.sqrt(mean_squared_error(model_lgbm.predict(Xtest),ytest))
```

5494.0777923513515

LightGBM's default hyperparameter settings also seem to be more robust as compared to those of XGBoost.

### 12.2.4 Tuning CatBoostRegressor

The CatBoost hyperparameters can be tuned just like the XGBoost hyperparameters. However, there is some difference in the hyperparameters of both the packages. For example, `reg_alpha` (the *L1 penalization on weights of leaves*) and `colsample_bytree` (*subsample ratio of columns when constructing each tree*) hyperparameters are not there in CatBoost.

```
#K-fold cross validation to find optimal parameters for CatBoost regressor
start_time = time.time()
param_grid = {'max_depth': [4,6,8, 10],
              'num_leaves': [20, 31, 40, 60],
              'learning_rate': [0.01, 0.05, 0.1],
              'reg_lambda': [0, 10, 100],
              'n_estimators': [500, 1000, 1500],
              'subsample': [0.5, 0.75, 1.0],
              'colsample_bylevel': [0.25, 0.5, 0.75, 1.0]}

cv = KFold(n_splits=5,shuffle=True,random_state=1)
optimal_params = RandomizedSearchCV(estimator=CatBoostRegressor(random_state=1, verbose=False),
                                   grow_policy='Lossguide'),
                                   param_distributions = param_grid, n_iter = 200,
                                   verbose = 1,random_state = 1, scoring='neg_root_mean_squared_er
```

```

        n_jobs=-1,
        cv = cv)

optimal_params.fit(X,y)
print("Optimal parameter values =", optimal_params.best_params_)
print("Optimal cross validation RMSE = ",optimal_params.best_score_)
print("Time taken = ", round((time.time()-start_time)/60), " minutes")

```

Fitting 5 folds for each of 200 candidates, totalling 1000 fits

Optimal parameter values = {'subsample': 0.5, 'reg\_lambda': 0, 'num\_leaves': 40, 'n\_estimators': 2000}

Optimal cross validation RMSE = -4993.129407810791

Time taken = 23 minutes

```
#RMSE based on the optimal parameter values
```

```
np.sqrt(mean_squared_error(optimal_params.best_estimator_.predict(Xtest),ytest))
```

5249.434282204398

It takes 2 minutes to tune CatBoost, which is higher than LightGBM and lesser than XGBoost. CatBoost falls in between LightGBM and XGBoost in terms of speed. However, it is likely to be more accurate than XGBoost and LightGBM, and likely to require lesser tuning as compared to XGBoost.

```
model = CatBoostRegressor(grow_policy='Lossguide')
```

```

grid = {'num_leaves': Integer(4, 64),
        'learning_rate': Real(0.0001, 1.0),
        'reg_lambda': Real(0, 1e4),
        'n_estimators': Integer(2, 2000),
        'subsample': Real(0.1, 1.0),
        'colsample_bylevel': Real(0.1, 1.0)}

```

```
kfold = KFold(n_splits = 5, shuffle = True, random_state = 1)
```

```

gcv = BayesSearchCV(model, search_spaces = grid, cv = kfold, n_iter = 200, random_state = 1,
                    scoring = 'neg_root_mean_squared_error', n_jobs = -1)

```

```

paras = list(gcv.search_spaces.keys())
paras.sort()

```

```

def monitor(optim_result):
    cv_values = pd.Series(optim_result['func_vals']).cummin()
    display.clear_output(wait = True)

```

```

min_ind = pd.Series(optim_result['func_vals']).argmin()
print(paras, "=", optim_result['x_iters'][min_ind], pd.Series(optim_result['func_vals'])
sns.lineplot(cv_values)
plt.show()
gcv.fit(X, y, callback = monitor)

```

<IPython.core.display.Image object>

```

# Optimal values obtained
#['colsample_bylevel', 'learning_rate', 'n_estimators', 'num_leaves', 'reg_lambda', 'subsamp
#[0.3745508446405472, 0.1000958551500621, 2000, 11, 0.0, 0.3877212027881348] 5132.5378396768

```

## 12.2.5 Tuning Tips

Check the [documentation](#) for some tuning tips.

1. It is not recommended to use values greater than 64 for `num_leaves`, since it can significantly slow down the training process.
2. In most cases, the optimal depth ranges from 4 to 10. Values in the range from 6 to 10 are recommended. The maximum possible value of `max_depth` is 16.
3. Do not use one-hot encoding during preprocessing. This affects both the training speed and the resulting quality.
4. Symmetric trees have a very good prediction speed (roughly 10 times faster than non-symmetric trees) and give better quality in many cases.

# 13 Ensemble modeling

Ensembling models can help reduce error by leveraging the diversity and collective wisdom of multiple models. When ensembling, several individual models are trained independently and their predictions are combined to make the final prediction.

We have already seen examples of ensemble models in chapters 5 - 13. The ensembled models may reduce error by reducing the bias (*boosting*) and / or reducing the variance (*bagging* / *random forests* / *boosting*).

However, in this chapter we'll ensemble different types of models, instead of the same type of model. We may ensemble a linear regression model, a random forest, a gradient boosting model, and as many different types of models as we wish.

Below are a couple of reasons why ensembling models can be effective in reducing error:

1. **Bias reduction:** Different models may have different biases and the ensemble can help mitigate the individual biases, leading to a more generalized and accurate prediction. For example, consider that one model has a positive bias, and another model has a negative bias for the same instance. By averaging or combining the predictions of the two models, the biases may cancel out.
2. **Variance reduction:** As seen in the case of random forests and bagged trees, by averaging or combining the predictions of multiple models, the ensemble can reduce the overall variance and improve the accuracy of the final prediction. Note that for variance reduction, the models should have a low correlation (*recall the variance reduction formula of random forests*).

Mathematically also, we can show the effectiveness of an ensemble model. Let's consider the case of regression, and let the predictors be denoted as  $X$ , and the response as  $Y$ . Let  $f_1, \dots, f_m$  be individual models. The expected MSE of an ensemble can be written as:

$$E(MSE_{Ensemble}) = E\left[\left(\frac{1}{m} \sum_{i=1}^m f_i(X) - Y\right)^2\right] = \frac{1}{m^2} \sum_{i=1}^m E\left[(f_i(X) - Y)^2\right] + \frac{1}{m^2} \sum_{i \neq j} E\left[(f_i(X) - Y)(f_j(X) - Y)\right]$$
$$\Rightarrow E(MSE_{Ensemble}) = \frac{1}{m} \left( \frac{1}{m} \sum_{i=1}^m E\left[(f_i(X) - Y)^2\right] \right) + \frac{1}{m^2} \sum_{i \neq j} E\left[(f_i(X) - Y)(f_j(X) - Y)\right]$$



$$\Rightarrow E(MSE_{Ensemble}) = \frac{1}{m} \left( \frac{1}{m} \sum_{i=1}^m E(MSE_{f_i}) \right) + \frac{1}{m^2} \sum_{i \neq j} E \left[ (f_i(X) - Y)(f_j(X) - Y) \right]$$

If  $f_1, \dots, f_m$  are unbiased, then,

$$E(MSE_{Ensemble}) = \frac{1}{m} \left( \frac{1}{m} \sum_{i=1}^m E(MSE_{f_i}) \right) + \frac{1}{m^2} \sum_{i \neq j} Cov(f_i(X), f_j(X))$$

Assuming the **models are uncorrelated** (*i.e., they have a zero correlation*), the second term (*covariance of  $f_i(\cdot)$  and  $f_j(\cdot)$* ) reduces to zero, and the expected MSE of the ensemble reduces to:

$$E(MSE_{Ensemble}) = \frac{1}{m} \left( \frac{1}{m} \sum_{i=1}^m E(MSE_{f_i}) \right) \quad (13.1)$$

Thus, the expected MSE of an ensemble model with uncorrelated models is much smaller than the average MSE of all the models. Unless there is a model that is much better than the rest of the models, the MSE of the ensemble model is likely to be lower than the MSE of the individual models. However, there is no guarantee that the MSE of the ensemble model will be lower than the MSE of the individual models. Consider an extreme case where only one of the models have a zero MSE. The MSE of this model will be lower than the expected MSE of the ensemble model.

```
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import cross_val_score, train_test_split, GridSearchCV, ParameterGrid,
StratifiedKFold, RandomizedSearchCV
from sklearn.metrics import mean_squared_error, r2_score, roc_curve, auc, precision_recall_curve
from sklearn.model_selection import KFold
from sklearn.tree import DecisionTreeRegressor, DecisionTreeClassifier
from sklearn.ensemble import VotingRegressor, VotingClassifier, StackingRegressor, \
StackingClassifier, GradientBoostingRegressor, GradientBoostingClassifier, BaggingRegressor, \
BaggingClassifier, RandomForestRegressor, RandomForestClassifier, AdaBoostRegressor, AdaBoostClassifier
from sklearn.linear_model import LinearRegression, LogisticRegression, LassoCV, RidgeCV, ElasticNetCV
from sklearn.neighbors import KNeighborsRegressor
import itertools as it
```

```
import time as time
import xgboost as xgb
from catboost import CatBoostRegressor
from lightgbm import LGBMRegressor
from sklearn.preprocessing import PolynomialFeatures
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
```

```
#Using the same datasets as used for linear regression in STAT303-2,
#so that we can compare the non-linear models with linear regression
trainf = pd.read_csv('./Datasets/Car_features_train.csv')
trainp = pd.read_csv('./Datasets/Car_prices_train.csv')
testf = pd.read_csv('./Datasets/Car_features_test.csv')
testp = pd.read_csv('./Datasets/Car_prices_test.csv')
train = pd.merge(trainf, trainp)
test = pd.merge(testf, testp)
train.head()
```

	carID	brand	model	year	transmission	mileage	fuelType	tax	mpg	engineSize	price
0	18473	bmw	6 Series	2020	Semi-Auto	11	Diesel	145	53.3282	3.0	37980
1	15064	bmw	6 Series	2019	Semi-Auto	10813	Diesel	145	53.0430	3.0	33980
2	18268	bmw	6 Series	2020	Semi-Auto	6	Diesel	145	53.4379	3.0	36850
3	18480	bmw	6 Series	2017	Semi-Auto	18895	Diesel	145	51.5140	3.0	25998
4	18492	bmw	6 Series	2015	Automatic	62953	Diesel	160	51.4903	3.0	18990

```
X = train[['mileage', 'mpg', 'year', 'engineSize']]
Xtest = test[['mileage', 'mpg', 'year', 'engineSize']]
y = train['price']
ytest = test['price']
```

## 13.1 Ensembling regression models

### 13.1.1 Voting Regressor

Here, we will combine the predictions of different models. The function `VotingRegressor()` averages the predictions of all the models.

Below are the individual models tuned in the previous chapters.

```

#Tuned AdaBoost model from Section 7.2.4
model_ada = AdaBoostRegressor(estimator=DecisionTreeRegressor(max_depth=10),n_estimators=50,
                             learning_rate=1.0, random_state=1).fit(X, y)
print("RMSE for AdaBoost = ", np.sqrt(mean_squared_error(model_ada.predict(Xtest), ytest)))

#Tuned Random forest model from Section 6.1.2
model_rf = RandomForestRegressor(n_estimators=300, random_state=1,
                                n_jobs=-1, max_features=2).fit(X, y)
print("RMSE for Random forest = ", np.sqrt(mean_squared_error(model_rf.predict(Xtest), ytest)))

# Tuned XGBoost model from Section 9.2.6
model_xgb = xgb.XGBRegressor(random_state=1,max_depth=8,n_estimators=1000, subsample = 0.75,
                             learning_rate = 0.01,reg_lambda=1, gamma = 100).fit(X, y)
print("RMSE for XGBoost = ", np.sqrt(mean_squared_error(model_xgb.predict(Xtest), ytest)))

#Tuned gradient boosting model from Section 8.2.5
model_gb = GradientBoostingRegressor(max_depth=8,n_estimators=100,learning_rate=0.1,
                                     random_state=1,loss='huber').fit(X, y)
print("RMSE for Gradient Boosting = ", np.sqrt(mean_squared_error(model_gb.predict(Xtest), ytest)))

# Tuned Light GBM model from Section 13.1.1
model_lgbm = LGBMRegressor(subsample = 0.5, reg_lambda = 0, reg_alpha = 100, boosting_type =
                            num_leaves = 31, n_estimators = 500, learning_rate = 0.05, colsample_bytree = 1.0,
                            top_rate = 0.5).fit(X, y)
print("RMSE for LightGBM = ", np.sqrt(mean_squared_error(model_lgbm.predict(Xtest), ytest)))

# Tuned CatBoost model from Section 13.2.3
model_cat = CatBoostRegressor(subsample=0.5, num_leaves=40, n_estimators=500, max_depth=10,
                              verbose = False, learning_rate = 0.05, colsample_bylevel=0.75,
                              grow_policy='Lossguide', random_state = 1).fit(X, y)
print("RMSE for CatBoost = ", np.sqrt(mean_squared_error(model_cat.predict(Xtest), ytest)))

```

```

RMSE for AdaBoost = 5693.165811600585
RMSE for Random forest = 5642.45839697972
RMSE for XGBoost = 5497.553788113875
RMSE for Gradient Boosting = 5405.787029062213
RMSE for LightGBM = 5355.964600884197
RMSE for CatBoost = 5271.104736146779

```

Note that we **don't need to fit** the models **individually** before fitting them simultaneously in the voting ensemble. If we fit them individual, it will unnecessarily **waste time**.

Let us ensemble the models using the voting ensemble with equal weights.

```
#Voting ensemble: Averaging the predictions of all models

#Tuned AdaBoost model from Section 7.2.4
model_ada = AdaBoostRegressor(estimator=DecisionTreeRegressor(max_depth=10),
                              n_estimators=50, learning_rate=1.0, random_state=1)

#Tuned Random forest model from Section 6.1.2
model_rf = RandomForestRegressor(n_estimators=300, random_state=1,
                                n_jobs=-1, max_features=2)

# Tuned XGBoost model from Section 9.2.6
model_xgb = xgb.XGBRegressor(random_state=1, max_depth=8, n_estimators=1000, subsample = 0.75,
                             colsample_bytree = 1.0, learning_rate = 0.01, reg_lambda=1, gamma = 100)

#Tuned gradient boosting model from Section 8.2.5
model_gb = GradientBoostingRegressor(max_depth=8, n_estimators=100, learning_rate=0.1,
                                     random_state=1, loss='huber')

# Tuned CatBoost model from Section 13.2.3
model_cat = CatBoostRegressor(subsample=0.5, num_leaves=40, n_estimators=500, max_depth=10,
                              learning_rate = 0.05, colsample_bylevel=0.75, grow_policy='Loss',
                              random_state=1, verbose = False)

# Tuned Light GBM model from Section 13.1.1
model_lgbm = LGBMRegressor(subsample = 0.5, reg_lambda = 0, reg_alpha = 100, boosting_type =
                           num_leaves = 31, n_estimators = 500, learning_rate = 0.05,
                           colsample_bytree = 1.0, top_rate = 0.5)

start_time = time.time()
en = VotingRegressor(estimators = [('xgb', model_xgb), ('ada', model_ada), ('rf', model_rf),
                                  ('gb', model_gb), ('cat', model_cat), ('lgbm', model_lgbm)], n_jobs = -1)
en.fit(X, y)
print("Ensemble model RMSE = ", np.sqrt(mean_squared_error(en.predict(Xtest), ytest)))
print("Time taken = ", np.round((time.time() - start_time)/60, 2), "minutes")
```

Ensemble model RMSE = 5259.899392611916

Time taken = 0.21 minutes

As expected, RMSE of the ensembled model is less than that of each of the individual models.

Note that the RMSE can be **further improved** by **removing** the **weaker models** from the ensemble. Let us remove the three weakest models - XGBoost, Random forest, and Adaboost.

```
#Voting ensemble: Averaging the predictions of all models

start_time = time.time()
en = VotingRegressor(estimators = [('gb',model_gb), ('cat', model_cat), ('lgbm', model_lgbm)])
en.fit(X,y)
print("Ensemble model RMSE = ", np.sqrt(mean_squared_error(en.predict(Xtest),ytest)))
print("Time taken = ", np.round((time.time() - start_time)/60,2), "minutes")
```

```
Ensemble model RMSE = 5191.814866810768
Time taken = 0.18 minutes
```

### 13.1.2 Stacking Regressor

Stacking is a more sophisticated method of ensembling models. The method is as follows:

1. The training data is split into  $K$  folds. Each of the  $K$  folds serves as a test data in one of the  $K$  iterations, and the rest of the folds serve as train data.
2. Each model is used to make predictions on each of the  $K$  folds, after being trained on the remaining  $K-1$  folds. In this manner, each model predicts the response on each train data point - when that train data point was not used to train the model.
3. Predictions at each training data points are generated by each model in step 2 (the above step). These predictions are now used as predictors to train a meta-model (referred by the argument `final_estimator`), with the original response as the response. The meta-model (or `final_estimator`) learns to combine predictions of different models to make a better prediction.

#### 13.1.2.1 Metamodel: Linear regression

```
#Stacking using LinearRegression as the metamodel
en = StackingRegressor(estimators = [('xgb', model_xgb), ('ada', model_ada), ('rf', model_rf),
                                   ('gb', model_gb), ('cat', model_cat), ('lgbm', model_lgbm)],
                      final_estimator=LinearRegression(),
                      cv = KFold(n_splits = 5, shuffle = True, random_state=1))
start_time = time.time()
en.fit(X,y)
```

```
print("Linear regression metamodel RMSE = ", np.sqrt(mean_squared_error(en.predict(Xtest), ytest)))
print("Time taken = ", np.round((time.time() - start_time)/60,2), "minutes")
```

```
Linear regression metamodel RMSE = 5220.456280327686
Time taken = 2.03 minutes
```

```
#Co-efficients of the meta-model
en.final_estimator_.coef_
```

```
array([ 0.05502964,  0.14566665,  0.01093624,  0.30478283,  0.57403909,
        -0.07057344])
```

```
sum(en.final_estimator_.coef_)
```

```
1.0198810182715363
```

Note the above coefficients of the meta-model. The model gives the **highest weight** to the **gradient boosting** model (*with **huber** loss*), and the **catboost** model, and the **lowest weight** to the relatively weak **random forest** model.

Also, note that the **coefficients need not sum to one**.

Let us try improving the RMSE further by removing the weaker models from the ensemble. Let us remove the three weakest models based on the size of their coefficients in the linear regression metamodel.

```
#Stacking using LinearRegression as the metamodel
en = StackingRegressor(estimators = [('gb', model_gb), ('cat', model_cat), ('ada', model_ada)],
                       final_estimator=LinearRegression(),
                       cv = KFold(n_splits = 5, shuffle = True, random_state=1))
start_time = time.time()
en.fit(X,y)
print("Linear regression metamodel RMSE = ", np.sqrt(mean_squared_error(en.predict(Xtest), ytest)))
print("Time taken = ", np.round((time.time() - start_time)/60,2), "minutes")
```

```
Linear regression metamodel RMSE = 5205.225710180056
Time taken = 1.36 minutes
```

The metamodel accuracy **improves further**, when **strong models** are ensembled.

```
#Co-efficients of the meta-model
en.final_estimator_.coef_
```

```
array([0.31824119, 0.54231032, 0.15998634])
```

```
sum(en.final_estimator_.coef_)
```

```
1.020537847948332
```

### 13.1.2.2 Metamodel: Lasso

```
#Stacking using Lasso as the metamodel
en = StackingRegressor(estimators = [('xgb', model_xgb), ('ada', model_ada), ('rf', model_rf),
                                   ('gb', model_gb), ('cat', model_cat), ('lgbm', model_lgbm) ],
                      final_estimator = LassoCV(),
                      cv = KFold(n_splits = 5, shuffle = True, random_state=1))
start_time = time.time()
en.fit(X,y)
print("Lasso metamodel RMSE = ", np.sqrt(mean_squared_error(en.predict(Xtest),ytest)))
print("Time taken = ", np.round((time.time() - start_time)/60,2), "minutes")
```

```
Lasso metamodel RMSE = 5206.021083501416
Time taken = 2.05 minutes
```

```
#Coefficients of the lasso metamodel
en.final_estimator_.coef_
```

```
array([ 0.03524446,  0.15077605,  0.          ,  0.30392268,  0.52946243,
        -0.          ])
```

Note that lasso **reduces the weight** of the **weak random forest** model, and **light gbm** model to **0**. Even though light GBM is a strong model, it may be **correlated or collinear** with XGBoost, or other models, and hence is not needed.

Note that as lasso performs **model selection** on its own, removing models with zero coefficients or weights does not make a difference, as shown below.

```
#Stacking using Lasso as the metamodel
en = StackingRegressor(estimators = [('xgb', model_xgb), ('ada', model_ada),
                                   ('gb', model_gb), ('cat', model_cat) ],
                      final_estimator = LassoCV(),
                      cv = KFold(n_splits = 5, shuffle = True, random_state=1))
start_time = time.time()
en.fit(X,y)
print("Lasso metamodel RMSE = ", np.sqrt(mean_squared_error(en.predict(Xtest),ytest)))
print("Time taken = ", np.round((time.time() - start_time)/60,2), "minutes")
```

```
Lasso metamodel RMSE = 5205.93233977352
Time taken = 1.79 minutes
```

```
#Coefficients of the lasso metamodel
en.final_estimator_.coef_
```

```
array([0.03415944, 0.15053122, 0.30464838, 0.53006297])
```

### 13.1.2.3 Metamodel: Random forest

A highly flexible model such as a random forest may not be a good choice for ensembling correlated models. However, let us tune the random forest meta model, and check its accuracy.

```
# Tuning hyperparameter of the random forest meta-model
start_time = time.time()
oob_score_i = []
for i in range(1, 7):
    en = StackingRegressor(estimators = [('xgb', model_xgb), ('ada', model_ada), ('rf', model_rf),
                                       ('gb', model_gb), ('cat', model_cat), ('lgbm', model_lgbm)],
                          final_estimator = RandomForestRegressor(max_features = i, oob_score = True),
                          cv = KFold(n_splits = 5, shuffle = True, random_state=1)).fit(X,y)
    oob_score_i.append(en.final_estimator_.oob_score_)
print("Time taken = ", np.round((time.time() - start_time)/60,2), "minutes")
```

```
Time taken = 12.08 minutes
```

```
print("Optimal value of max_features =", np.array(oob_score_i).argmax() + 1)
```

```
Optimal value of max_features = 1
```



```
# Training the tuned random forest metamodel
start_time = time.time()
en = StackingRegressor(estimators = [('xgb', model_xgb), ('ada', model_ada),
                                   ('rf', model_rf), ('gb', model_gb), ('cat', model_cat),
                                   ('lgbm', model_lgbm)],
                      final_estimator = RandomForestRegressor(max_features = 1,
                                                              n_estimators=500), cv = KFold(n_splits = 5, shuffle = True,
                                                              random_state=1)).fit(X,y)
print("Random Forest metamodel RMSE = ", np.sqrt(mean_squared_error(en.predict(Xtest),ytest)))
print("Time taken = ", np.round((time.time() - start_time)/60,2), "minutes")
```

```
Random Forest metamodel RMSE = 5441.9155087961
Time taken = 1.71 minutes
```

Note that highly flexible models may not be needed when the predictors are highly correlated with the response. However, in some cases, they may be useful, as in the classification example in the next section.

#### 13.1.2.4 Metamodel: CatBoost

```
#Stacking using MARS as the meta-model
en = StackingRegressor(estimators = [('xgb', model_xgb), ('ada', model_ada), ('rf', model_rf),
                                   ('gb', model_gb), ('cat', model_cat), ('lgbm', model_lgbm)],
                      final_estimator = CatBoostRegressor(verbose = False),
                      cv = KFold(n_splits = 5, shuffle = True, random_state=1))
start_time = time.time()
en.fit(X,y)
print("Random Forest metamodel RMSE = ", np.sqrt(mean_squared_error(en.predict(Xtest),ytest)))
print("Time taken = ", np.round((time.time() - start_time)/60,2), "minutes")
```

```
Random Forest metamodel RMSE = 5828.803609683251
Time taken = 1.66 minutes
```

## 13.2 Ensembling classification models

We'll ensemble models for predicting accuracy of identifying people having a heart disease.

```

data = pd.read_csv('./Datasets/Heart.csv')
data.dropna(inplace = True)
#Response variable
y = pd.get_dummies(data['AHD'])['Yes']

#Creating a dataframe for predictors with dummy variables replacing the categorical variables
X = data.drop(columns = ['AHD','ChestPain','Thal'])
X = pd.concat([X,pd.get_dummies(data['ChestPain']),pd.get_dummies(data['Thal'])],axis=1)

#Creating train and test datasets
Xtrain,Xtest,ytrain,ytest = train_test_split(X,y,train_size = 0.5,random_state=1)

```

Let us tune the individual models first.

## AdaBoost

```

# Tuning Adaboost for maximizing accuracy
model = AdaBoostClassifier(random_state=1)
grid = dict()
grid['n_estimators'] = [10, 50, 100,200,500]
grid['learning_rate'] = [0.0001, 0.001, 0.01,0.1, 1.0]
grid['base_estimator'] = [DecisionTreeClassifier(max_depth=1), DecisionTreeClassifier(max_depth=2),
                           DecisionTreeClassifier(max_depth=3),DecisionTreeClassifier(max_depth=4)]
# define the evaluation procedure
cv = StratifiedKFold(n_splits=5, shuffle=True, random_state=1)
# define the grid search procedure
grid_search = GridSearchCV(estimator=model, param_grid=grid, n_jobs=-1, cv=cv, scoring='accuracy')
# execute the grid search
grid_result = grid_search.fit(Xtrain, ytrain)
# summarize the best score and configuration
print("Best: %f using %s" % (grid_result.best_score_, grid_result.best_params_))

```

Best: 0.871494 using {'base\_estimator': DecisionTreeClassifier(max\_depth=1), 'learning\_rate': 0.0001}

## Gradient Boosting

```

# Tuning gradient boosting for maximizing accuracy
model = GradientBoostingClassifier(random_state=1)
grid = dict()
grid['n_estimators'] = [10, 50, 100,200,500]
grid['learning_rate'] = [0.0001, 0.001, 0.01,0.1, 1.0]
grid['max_depth'] = [1,2,3,4,5]
grid['subsample'] = [0.5,1.0]
# define the evaluation procedure
cv = StratifiedKFold(n_splits=5, shuffle=True, random_state=1)
# define the grid search procedure
grid_search = GridSearchCV(estimator=model, param_grid=grid, n_jobs=-1, cv=cv, scoring='accuracy')
# execute the grid search
grid_result = grid_search.fit(Xtrain, ytrain)
# summarize the best score and configuration
print("Best: %f using %s" % (grid_result.best_score_, grid_result.best_params_))

```

Best: 0.871954 using {'learning\_rate': 1.0, 'max\_depth': 4, 'n\_estimators': 100, 'subsample': 0.5}

## XGBoost

```

# Tuning XGBoost for maximizing accuracy
start_time = time.time()
param_grid = {'n_estimators': [25, 100,250,500],
              'max_depth': [4, 6 ,8],
              'learning_rate': [0.01,0.1,0.2],
              'gamma': [0, 1, 10, 100],
              'reg_lambda': [0, 10, 100],
              'subsample': [0.5, 0.75, 1.0]
              'scale_pos_weight': [1.25,1.5,1.75]}#Control the balance of positive and negative samples

cv = StratifiedKFold(n_splits=5,shuffle=True,random_state=1)
optimal_params = GridSearchCV(estimator=xgb.XGBClassifier(random_state=1),
                              param_grid = param_grid,
                              scoring = 'accuracy',
                              verbose = 1,
                              n_jobs=-1,
                              cv = cv)
optimal_params.fit(Xtrain,ytrain)

```

```
print(optimal_params.best_params_,optimal_params.best_score_)
print("Time taken = ", (time.time()-start_time)/60, " minutes")
```

Fitting 5 folds for each of 972 candidates, totalling 4860 fits

```
{'gamma': 0, 'learning_rate': 0.2, 'max_depth': 4, 'n_estimators': 25, 'reg_lambda': 0, 'scale_pos_weight': 1.25}
```

Time taken = 0.9524135629336039 minutes

```
#Tuned Adaboost model
model_ada = AdaBoostClassifier(base_estimator=DecisionTreeClassifier(max_depth=1), n_estimators=100,
                               random_state=1, learning_rate=0.01).fit(Xtrain, ytrain)
test_accuracy_ada = model_ada.score(Xtest, ytest) #Returns the classification accuracy of the model

#Tuned Random forest model from Section 6.3
model_rf = RandomForestClassifier(n_estimators=500, random_state=1, max_features=3,
                                 n_jobs=-1, oob_score=False).fit(Xtrain, ytrain)
test_accuracy_rf = model_rf.score(Xtest, ytest) #Returns the classification accuracy of the model

#Tuned gradient boosting model
model_gb = GradientBoostingClassifier(n_estimators=100, random_state=1, max_depth=4, learning_rate=0.1,
                                      subsample = 1.0).fit(Xtrain, ytrain)
test_accuracy_gb = model_gb.score(Xtest, ytest) #Returns the classification accuracy of the model

#Tuned XGBoost model
model_xgb = xgb.XGBClassifier(random_state=1, gamma=0, learning_rate = 0.2, max_depth=4,
                              n_estimators = 25, reg_lambda = 0, scale_pos_weight=1.25).fit(Xtrain, ytrain)
test_accuracy_xgb = model_xgb.score(Xtest, ytest) #Returns the classification accuracy of the model

print("Adaboost accuracy = ", test_accuracy_ada)
print("Random forest accuracy = ", test_accuracy_rf)
print("Gradient boost accuracy = ", test_accuracy_gb)
print("XGBoost model accuracy = ", test_accuracy_xgb)
```

```
Adaboost accuracy = 0.7986577181208053
Random forest accuracy = 0.8120805369127517
Gradient boost accuracy = 0.7986577181208053
XGBoost model accuracy = 0.7785234899328859
```

### 13.2.1 Voting classifier - hard voting

In this type of ensembling, the predicted class is the one predicted by the majority of the classifiers.

```
ensemble_model = VotingClassifier(estimators=[('ada',model_ada),('rf',model_rf),('gb',model_gb)],
ensemble_model.fit(Xtrain,ytrain)
ensemble_model.score(Xtest, ytest)
```

0.825503355704698

Note that the prediction accuracy of the ensemble is higher than the prediction accuracy of each of the individual models on unseen data.

### 13.2.2 Voting classifier - soft voting

In this type of ensembling, the predicted class is the one based on the average predicted probabilities of all the classifiers. The threshold probability is 0.5.

```
ensemble_model = VotingClassifier(estimators=[('ada',model_ada),('rf',model_rf),('gb',model_gb)],
                                voting='soft')
ensemble_model.fit(Xtrain,ytrain)
ensemble_model.score(Xtest, ytest)
```

0.7919463087248322

Note that soft voting will be good only for well calibrated classifiers, i.e., all the classifiers must have probabilities at the same scale.

### 13.2.3 Stacking classifier

Conceptually, the idea is similar to that of Stacking regressor.

```
#Using Logistic regression as the meta model (final_estimator)
ensemble_model = StackingClassifier(estimators=[('ada',model_ada),('rf',model_rf),('gb',model_gb)],
                                final_estimator=LogisticRegression(random_state=1,max_iter=1000),
                                cv = StratifiedKFold(n_splits=5,shuffle=True,random_state=1))
ensemble_model.fit(Xtrain,ytrain)
ensemble_model.score(Xtest, ytest)
```

0.7986577181208053

```
#Coefficients of the logistic regression metamodel
ensemble_model.final_estimator_.coef_
```

```
array([[0.81748051, 1.28663164, 1.64593342, 1.50947087]])
```

```
#Using random forests as the meta model (final_estimator). Note that random forest will require tuning
ensemble_model = StackingClassifier(estimators=[('ada',model_ada),('rf',model_rf),('gb',model_gb)],
                                   final_estimator=RandomForestClassifier(n_estimators=500, n_jobs=-1,
                                                                           random_state=1,oob_score=True),
                                   cv = StratifiedKFold(n_splits=5,shuffle=True,random_state=1))
ensemble_model.fit(Xtrain,ytrain)
ensemble_model.score(Xtest, ytest)
```

```
0.8322147651006712
```

Note that a complex `final_estimator` such as random forest will require tuning. In the above case, the `max_features` argument of random forests has been tuned to obtain the maximum OOB score. The tuning is shown below.

```
#Tuning the random forest parameters
start_time = time.time()
oob_score = {}

i=0
for pr in range(1,5):
    model = StackingClassifier(estimators=[('ada',model_ada),('rf',model_rf),('gb',model_gb)],
                              final_estimator=RandomForestClassifier(n_estimators=500, n_jobs=-1,
                                                                      random_state=1,oob_score=True),
                              cv = StratifiedKFold(n_splits=5,shuffle=True,random_state=1))
    oob_score[pr] = model.final_estimator_.oob_score_

end_time = time.time()
print("time taken = ", (end_time-start_time)/60, " minutes")
print("max accuracy = ", np.max(list(oob_score.values())))
print("Best value of max_features= ", np.argmax(list(oob_score.values()))+1)
```

```
time taken = 0.33713538646698 minutes
max accuracy = 0.8445945945945946
Best value of max_features= 1
```

```
#The final predictor (metamodel) - random forest obtains the maximum oob_score for max_features
oob_score
```

```
{1: 0.8445945945945946,
 2: 0.831081081081081,
 3: 0.8378378378378378,
 4: 0.831081081081081}
```

### 13.2.4 Tuning all models simultaneously

Individual model hyperparameters can be tuned simultaneously while ensembling them with a `VotingClassifier()`. However, this approach can be too expensive for even moderately-sized datasets.

```
# Create the param grid with the names of the models as prefixes

model_ada = AdaBoostClassifier(base_estimator = DecisionTreeClassifier())
model_rf = RandomForestClassifier()
model_gb = GradientBoostingClassifier()
model_xgb = xgb.XGBClassifier()

ensemble_model = VotingClassifier(estimators=[('ada',model_ada),('rf',model_rf),('gb',model_gb)])

hp_grid = dict()

# XGBoost
hp_grid['xgb__n_estimators'] = [25, 100,250,50]
hp_grid['xgb__max_depth'] = [4, 6 ,8]
hp_grid['xgb__learning_rate'] = [0.01, 0.1, 1.0]
hp_grid['xgb__gamma'] = [0, 1, 10, 100]
hp_grid['xgb__reg_lambda'] = [0, 1, 10, 100]
hp_grid['xgb__subsample'] = [0, 1, 10, 100]
hp_grid['xgb__scale_pos_weight'] = [1.0, 1.25, 1.5]
hp_grid['xgb__colsample_bytree'] = [0.5, 0.75, 1.0]

# AdaBoost
hp_grid['ada__n_estimators'] = [10, 50, 100,200,500]
hp_grid['ada__base_estimator__max_depth'] = [1, 3, 5]
hp_grid['ada__learning_rate'] = [0.01, 0.1, 0.2]

# Random Forest
```

```

hp_grid['rf__n_estimators'] = [100]
hp_grid['rf__max_features'] = [3, 6, 9, 12, 15]

# GradBoost
hp_grid['gb__n_estimators'] = [10, 50, 100, 200, 500]
hp_grid['gb__max_depth'] = [1, 3, 5]
hp_grid['gb__learning_rate'] = [0.01, 0.1, 0.2, 1.0]
hp_grid['gb__subsample'] = [0.5, 0.75, 1.0]

start_time = time.time()
grid = RandomizedSearchCV(ensemble_model, hp_grid, cv=5, scoring='accuracy', verbose = True,
                          n_iter = 100, n_jobs=-1).fit(Xtrain, ytrain)
print("Time taken = ", round((time.time()-start_time)/60), " minutes")

grid.best_estimator_.score(Xtest, ytest)

```

0.8120805369127517

### 13.3 Ensembling models based on different sets of predictors

Generally, tree-based models such as CatBoost, and XGBoost are the most accurate, while other models, such as bagging, random forests, KNN, and linear models, may not be as accurate. Thus, sometimes, the weaker models, despite bringing-in diversity in the model ensemble may deteriorate the ensemble accuracy due to their poor individual performance (*check slides for technical details*). Thus, sometimes, another approach to bring-in model diversity is to develop strong models based on different sets of predictors, and ensemble them.

Different feature selection methods (*such as Lasso, feature importance returned by tree-based methods, stepwise k-fold feature selection, etc.*), may be used to obtain different sets of important features, strong models can be tuned on these sets, and then ensembled. Even though the models may be of the same type, the different sets of predictors will help bring-in the element of diversity in the ensemble.

```

trainf = pd.read_csv('./Datasets/Car_features_train.csv')
trainp = pd.read_csv('./Datasets/Car_prices_train.csv')
testf = pd.read_csv('./Datasets/Car_features_test.csv')
testp = pd.read_csv('./Datasets/Car_prices_test.csv')
train = pd.merge(trainf, trainp)
test = pd.merge(testf, testp)
train.head()

```



	carID	brand	model	year	transmission	mileage	fuelType	tax	mpg	engineSize	price
0	18473	bmw	6 Series	2020	Semi-Auto	11	Diesel	145	53.3282	3.0	37980
1	15064	bmw	6 Series	2019	Semi-Auto	10813	Diesel	145	53.0430	3.0	33980
2	18268	bmw	6 Series	2020	Semi-Auto	6	Diesel	145	53.4379	3.0	36850
3	18480	bmw	6 Series	2017	Semi-Auto	18895	Diesel	145	51.5140	3.0	25998
4	18492	bmw	6 Series	2015	Automatic	62953	Diesel	160	51.4903	3.0	18990

```
X = train[['mileage','mpg','year','engineSize']]
Xtest = test[['mileage','mpg','year','engineSize']]
y = train['price']
ytest = test['price']
```

We will create polynomial interactions to develop two sets of predictors - first order predictors, and second order predictors.

```
poly_set = PolynomialFeatures(2, include_bias = False)
X_poly = poly_set.fit_transform(X)
X_poly = pd.DataFrame(X_poly, columns=poly_set.get_feature_names_out())
X_poly.columns = X_poly.columns.str.replace("^", "_", regex=True)
Xtest_poly = poly_set.fit_transform(Xtest)
Xtest_poly = pd.DataFrame(Xtest_poly, columns=poly_set.get_feature_names_out())
Xtest_poly.columns = Xtest_poly.columns.str.replace("^", "_", regex=True)
```

Let us use 2 different sets of predictors to introduce diversity in the ensemble.

```
col_set1 = ['mileage','mpg', 'year','engineSize']
col_set2 = X_poly.columns
```

Let us use two types of strong tree-based models.

```
cat = CatBoostRegressor(verbose=False)
gb = GradientBoostingRegressor(loss = 'huber')
```

We will use the `Pipeline()` function along with `ColumnTransformer()` to map a predictor set to each model.

```
cat_pipe1 = Pipeline([
    ('column_transformer', ColumnTransformer([('cat1_transform', 'passthrough', col_set1)],
    ('cat1', cat)
```

```

])

cat_pipe2 = Pipeline([
    ('column_transformer', ColumnTransformer([('cat2_transform', 'passthrough', col_set2)], r
    ('cat2', cat)
])

gb_pipe1 = Pipeline([
    ('column_transformer', ColumnTransformer([('gb1_transform', 'passthrough', col_set1)], r
    ('gb1', gb)
])

gb_pipe2 = Pipeline([
    ('column_transformer', ColumnTransformer([('gb2_transform', 'passthrough', col_set2)], r
    ('gb2', gb)
])

```

We will use Linear regression to ensemble the models.

```
en_new.final_estimator_.coef_
```

```
array([ 0.30127482,  0.79242981, -0.07168258, -0.01781781])
```

```

en_new = StackingRegressor(estimators = [('cat1', cat_pipe1), ('cat2', cat_pipe2),
                                       ('gb1', gb_pipe1), ('gb2', gb_pipe2)],
                           final_estimator=LinearRegression(),
                           cv = KFold(n_splits = 15, shuffle = True, random_state=1))

```

```
en_new.fit(X_poly, y)
```

```

StackingRegressor(cv=KFold(n_splits=15, random_state=1, shuffle=True),
                  estimators=[('cat1',
                               Pipeline(steps=[('column_transformer',
                                                  ColumnTransformer(transformers=[('cat1_transf
                                                  'passthrough',
                                                  ['mileage',
                                                  'mpg',
                                                  'year',
                                                  'engineSize
                               ('cat1',

```

```

<catboost.core.CatBoostRegressor object at 0x...
('cat2',
 Pipeline(steps=[('column_transformer',...
 Pipeline(steps=[('column_transformer',
                  ColumnTransformer(transformers=[('gb2_transf
                  'passthrough',
                  Index(['mileage year', 'mileage engineSize', 'mpg_2', 'mpg year',
                  'mpg engineSize', 'year_2', 'year engineSize', 'engineSize_2'],
dtype='object'))])),
                  ('gb2',
                   GradientBoostingRegressor(loss='huber'))]))]
final_estimator=LinearRegression())

```

```
mean_squared_error(en_new.predict(Xtest_poly), ytest, squared = False)
```

```
5185.376722607323
```

Note that the above model does better on test data than all the models developed so far. Using different sets of predictors introduces diversity in the ensemble, as an alternative to including “weaker” models in the ensemble to add diversity.

Check the idea being used in the Spring 2023 prediction problem in the appendix.

# A Assignment 1

## Instructions

1. You may talk to a friend, discuss the questions and potential directions for solving them. However, you need to write your own solutions and code separately, and not as a group activity.
2. Write your code in the **Code cells** and your answers in the **Markdown cells** of the Jupyter notebook. Ensure that the solution is written neatly enough to for the graders to understand and follow.
3. Use [Quarto](#) to render the **.ipynb** file as HTML. You will need to open the command prompt, navigate to the directory containing the file, and use the command: `quarto render filename.ipynb --to html`. Submit the HTML file.
4. The assignment is worth 100 points, and is due on **Thursday, 11th April 2025 at 11:59 pm**.
5. **Five points are properly formatting the assignment.** The breakdown is as follows:
  - Must be an HTML file rendered using Quarto (**2 points**). *If you have a Quarto issue, you must mention the issue & quote the error you get when rendering using Quarto in the comments section of Canvas, and submit the ipynb file.*
  - There aren't excessively long outputs of extraneous information (e.g. no printouts of entire data frames without good reason, there aren't long printouts of which iteration a loop is on, there aren't long sections of commented-out code, etc.) (**1 point**)
  - Final answers to each question are written in the Markdown cells. (**1 point**)
  - There is no piece of unnecessary / redundant code, and no unnecessary / redundant text. (**1 point**)

## A.1 1) Bias-Variance Trade-off for Regression (50 points)

The main goal of this question is to understand and visualize the bias-variance trade-off in a regression model by performing repetitive simulations.

The conceptual clarity about bias and variance will help with the main logic behind creating many models that will come up later in the course.

### A.1.1 a) Define the True Relationship (Signal)

First, you need to implement the underlying true relationship (Signal) you want to sample data from. Assume that the function is the [Bukin function](#). Implement it as a user-defined function and run it with the test cases below to make sure it is implemented correctly. **(5 points)**

**Note:** It would be more useful to have only one input to the function. You can treat the input as an array of two elements.

```
print(Bukin(np.array([1,2]))) # The output should be 141.177
print(Bukin(np.array([6,-4]))) # The output should be 208.966
print(Bukin(np.array([0,1]))) # The output should be 100.1
```

### A.1.2 b) Generate Test Set (No Noise)

Generate a **noiseless** test set with **100 observations** sampled from the true underlying function. This test set will be used to evaluate **bias and variance**, so make sure it follows the correct data generation process.

**(5 points)**

**Instructions:**

- **Do not use loops** for this question.
- `.apply` will be especially helpful (and often simpler).

**Data generation assumptions:**

- Use `np.random.seed(100)` for reproducibility.
- The first predictor,  $x_1$ , should be drawn from a **uniform distribution** over the interval  $[-15, -5]$ , i.e.,  $x_1 \sim U[-15, -5]$ .
- The second predictor,  $x_2$ , should be drawn from a **uniform distribution** over the interval  $[-3, 3]$ , i.e.,  $x_2 \sim U[-3, 3]$ .
- Compute the true function values using the underlying model as your response  $y$

### A.1.3 c) Initialize Results DataFrame

Create an empty DataFrame with the following columns: **degree**, **bias\_sq**, **var**, and **mse**. You will use this DataFrame to store the results of your bias-variance tradeoff analysis, which will be used for generating plots and interpreting model performance.

(3 points)

### A.1.4 d) Generate Training Sets (With Noise)

To estimate the **bias** and **variance** of a Linear Regression model fitted on noisy data from the underlying Bukin function, follow these steps:

#### Generate 100 Training Sets:

- Create **100 independent training datasets**, each containing **100 observations** (same size as the test set).
- For each training set:
  - Use `np.random.seed(i)` to ensure reproducibility, where `i` is the dataset index (0 to 99).
  - Draw predictors from the same distributions as in the test set.
  - Add **Gaussian noise** with mean 0 and **standard deviation 10**, i.e.,  $\mathcal{N}(0, 10)$ , to the target values.

#### Train Polynomial Models (Degrees 1 to 7):

- For each training set, train polynomial models of degrees **1 through 7**.
- Use polynomial feature transformations that include both:
  - **Higher-order terms** (e.g.,  $x_1^2$ ,  $x_2^3$ )
  - **Interaction terms** (e.g.,  $x_1 \cdot x_2$ )
- Use the **same fixed, noiseless test set** to generate predictions from each model.

#### Estimate Bias<sup>2</sup>, Variance, and MSE for Each Degree:

- For each **test point**, you will obtain **100 predicted values** (one from each trained model).
- Use these predictions to compute the following quantities **per degree** (averaged over all test points):
  - **Bias squared**
  - **Variance**
  - **Mean Squared Error (MSE)**
- Store these values in your results DataFrame for plotting and analysis.

(25 points)

### A.1.5 Reminder: Computing MSE on the Noiseless Test Set

When computing the **MSE**, you should **not** include the irreducible error (noise variance). Here's why:

- The **irreducible error** ( $\sigma^2 = 100$ ) comes from the noise **in the training data**.
- The **test set is noiseless**, so your predictions are evaluated against the **true function values**.
- Therefore:

$$\text{\$ MSE} = \text{Bias}^2 + \text{Variance} \text{\$}$$

#### Conclusion:

The MSE you compute is the **expected prediction error** on noiseless test data.

You should **only sum the estimated bias<sup>2</sup> and variance** — no additional noise term is needed.

### A.1.6 e) Visualize Bias-Variance Tradeoff

Using the results stored in your summary DataFrame, create a line plot showing how the following quantities change with the **degree** of the polynomial model:

- bias squared
- variance
- mse — computed as the sum of bias squared and variance

(5 points)

**Plot requirements:**

- The x-axis should represent the **polynomial degree**.
- The y-axis should represent the **error values**.
- Include a **legend** that clearly labels each line: bias<sup>2</sup>, variance, and mse.

(2 points)

### A.1.7 f) Identify the Optimal Model

- What is the **optimal polynomial degree** based on the lowest mean squared error (mse)?  
(2 points)
- Report the corresponding values of:
  - **bias squared**
  - **variance**
  - **mean squared error (mse)**  
for that optimal degree.  
(3 points)

## A.2 2) Building a Low-Bias-Low-Variance Model via Regularization (50 points)

The main goal of this question is to further reduce the **total error** by applying **regularization**. Specifically, you'll implement a **low-bias, low-variance** model for the underlying function and the noisy data.

### A.2.1 a) Why Regularization?

Explain why the model with the optimal degree (as identified in Question 1) is **not guaranteed** to be the true low-bias-low-variance model.

Why might **regularization** still be necessary to improve generalization performance?

(5 points)

### A.2.2 b) Which Degrees to Exclude?

Before repeating the process from Question 1, carefully examine the plot and results from **1e** and **1f**. Identify which polynomial degrees should be **excluded** from regularization experiments because they are already too simple or too complex.

Explain which degrees you will exclude and **why**, considering how **regularization affects bias and variance**.

(10 points)



### A.2.3 c) Apply Ridge Regularization

Repeat the steps from **1c** and **1d**, but this time using **Ridge regression** instead of ordinary least squares.

- **Exclude** the degrees you identified in **2b**, as well as **degree 7** (to avoid extreme over-fitting).
- Use **5-fold cross-validation** to tune the regularization hyperparameter.
- Use `neg_root_mean_squared_error` as the scoring metric for cross-validation.
- Try regularization strengths in the range `[1, 100]`.

(10 points)

### A.2.4 d) Visualize Regularized Results

Repeat the visualization from **1e**, but using the results obtained from **2c** (Ridge regression).

Plot: - **bias squared** - **variance** - **mse**

against the polynomial **degree**, using regularized models.

(10 points)

### A.2.5 e) Evaluate the Regularized Model

- What is the **degree** of the optimal Ridge Regression model based on the lowest total error?  
(3 points)

- What are the corresponding values of:

- **bias-squared**

- **variance**

- **mse**

- for that optimal Ridge model?

- (3 points)

- Compare these results to those of the optimal **Linear Regression** model from Question 1.

How did **regularization** affect the bias, variance, and total error?

(4 points)

### A.2.6 f) Interpreting the Impact of Regularization

- Was the **regularization successful** in reducing the **total error** of the regression model?  
(2 points)
- Based on the results from **2e**, explain how **bias** and **variance** changed with regularization.  
How did these changes contribute to the final total error?  
(3 points)

## **B Datasets, assignment and project files**

Datasets used in the book, assignment files, project files, and prediction problems report template can be found [here](#)