Data Science III with python (Class notes)

STAT 303-3

Arvind Krishna, Emre Besler, and Lizhen Shi

2023-03-24

Table of contents

Preface

These are class notes for the course STAT303-3. This is not the course text-book. You are required to read the relevant sections of the book as mentioned on the course website.

The course notes are currently being written, and will continue to being developed as the course progresses (just like the class notes last quarter). Please report any typos / mistakes / inconsistencies / issues with the class notes / class presentations in your comments here. Thank you!

Part I

Moving towards non-linearity

1 Introduction to scikit-learn

In this chapter, we'll learn some functions from the library `sklearn` that will be useful in:

1. Splitting the data into `train` and `test`
2. Scaling data
3. Fitting a model
4. Computing model performance metrics
5. Tuning model hyperparameters* to optimize the desired performance metric

**In machine learning, a model hyperparameter is a parameter that cannot be learned from training data and must be set before training the model. Hyperparameters control aspects of the model's behavior and can greatly impact its performance. For example, the regularization parameter λ, in linear regression is a hyperparameter. You need to specify it before fitting the model. On the other hand, the beta coefficients in linear regression are parameters, as you learn them while training the model, and don't need to specify their values beforehand.*

We'll use a classification problem to illustrate the functions. However, similar functions can be used for regression problems, i.e., prediction problems with a continuous response.

```
# 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)
```

Let us import the `sklearn` modules useful in developing statistical models.

```
# sklearn has 100s of models - grouped in sublibraries, such as linear_model
from sklearn.linear_model import LogisticRegression, LinearRegression

# sklearn has many tools for cleaning/processing data, also grouped in sublibraries
# splitting one dataset into train and test, computing cross validation score, cross valid
from sklearn.model_selection import train_test_split, cross_val_predict, cross_val_score
```

```
#sklearn module for scaling data
from sklearn.preprocessing import StandardScaler

#sklearn modules for computing the performance metrics
from sklearn.metrics import accuracy_score, mean_absolute_error, mean_squared_error, r2_sc
roc_curve, auc, precision_score, recall_score, confusion_matrix
```

```
#Reading data
data = pd.read_csv('./Datasets/diabetes.csv')
```

Scikit-learn doesn't support the formula-like syntax of specifying the response and the predictors as in the `statsmodels` library. We need to create separate objects for predictors and response, which should be *array-like*. A Pandas DataFrame / Series or a Numpy array are *array-like* objects.

Let us reference our predictors as object `X`, and the response as object `y`.

```
# Separating the predictors and response - THIS IS HOW ALL SKLEARN OBJECTS ACCEPT DATA (di
y = data.Outcome
X = data.drop("Outcome", axis = 1)
```

1.1 Splitting data into `train` and `test`

Let us create train and test datasets for developing a model to predict if a person has diabetes.

```
# Creating training and test data
    # 80-20 split, which is usual - 70-30 split is also fine, 90-10 is fine if the dataset
    # random_state to set a random seed for the splitting - reproducible results
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2, random_state =
```

Let us find the proportion of classes (*'having diabetes' ($y = 1$) or 'not having diabetes' ($y = 0$)*) in the complete dataset.

```
#Proportion of 0s and 1s in the complete data
y.value_counts()/y.shape
```

```
0    0.651042
1    0.348958
Name: Outcome, dtype: float64
```

Let us find the proportion of classes (*'having diabetes' ($y = 1$) or 'not having diabetes' ($y = 0$)*) in the train dataset.

```
#Proportion of 0s and 1s in train data
y_train.value_counts()/y_train.shape
```

```
0    0.644951
1    0.355049
Name: Outcome, dtype: float64
```

```
#Proportion of 0s and 1s in test data
y_test.value_counts()/y_test.shape
```

```
0    0.675325
1    0.324675
Name: Outcome, dtype: float64
```

We observe that the proportion of 0s and 1s in the `train` and `test` dataset are slightly different from that in the complete `data`. In order for these datasets to be more representative of the population, they should have a proportion of 0s and 1s similar to that in the complete dataset. This is especially critical in case of imbalanced datasets, where one class is represented by a significantly smaller number of instances than the other(s).

When training a classification model on an imbalanced dataset, the model might not learn enough about the minority class, which can lead to poor generalization performance on new data. This happens because the model is biased towards the majority class, and it might even predict all instances as belonging to the majority class.

1.1.1 Stratified splitting

We will use the argument `stratify` to obtain a proportion of 0s and 1s in the `train` and `test` datasets that is similar to the proportion in the complete 'data.

```
#Stratified train-test split
X_train_stratified, X_test_stratified, y_train_stratified,\
y_test_stratified = train_test_split(X, y, test_size = 0.2, random_state = 45, stratify=y)
```

```
#Proportion of 0s and 1s in train data with stratified split
y_train_stratified.value_counts()/y_train.shape
```

```
0    0.651466
1    0.348534
Name: Outcome, dtype: float64
```

```
#Proportion of 0s and 1s in test data with stratified split
y_test_stratified.value_counts()/y_test.shape
```

```
0    0.649351
1    0.350649
Name: Outcome, dtype: float64
```

The proportion of the classes in the stratified split mimics the proportion in the complete dataset more closely.

By using stratified splitting, we ensure that both the `train` and `test` data sets have the same proportion of instances from each class, which means that the model will see enough instances from the minority class during training. This, in turn, helps the model learn to distinguish between the classes better, leading to better performance on new data.

Thus, stratified splitting helps to ensure that the model sees enough instances from each class during training, which can improve the model's ability to generalize to new data, particularly in cases where one class is underrepresented in the dataset.

Let us develop a logistic regression model for predicting if a person has diabetes.

1.2 Scaling data

In certain models, it may be important to scale data for various reasons. In a logistic regression model, scaling can help with model convergence. Scikit-learn uses a method known as gradient-descent *(not in scope of the syllabus of this course)* to obtain a solution. In case the predictors have different orders of magnitude, the algorithm may fail to converge. In such cases, it is useful to standardize the predictors so that all of them are at the same scale.

```
# With linear/logistic regression in scikit-learn, especially when the predictors have dif
# of magn., scaling is necessary. This is to enable the training algo. which we did not co
scaler = StandardScaler().fit(X_train)
X_train_scaled = scaler.transform(X_train)
X_test_scaled = scaler.transform(X_test) # Do NOT refit the scaler with the test data, jus
```

1.3 Fitting a model

Let us fit a logistic regression model for predicting if a person has diabetes. Let us try fitting a model with the un-scaled data.

```
# Create a model object - not trained yet
logreg = LogisticRegression()

# Train the model
logreg.fit(X_train, y_train)
```

```
C:\Users\akl0407\Anaconda3\lib\site-packages\sklearn\linear_model\_logistic.py:763: Converge
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression
  n_iter_i = _check_optimize_result(
```

```
LogisticRegression()
```

Note that the model with the un-scaled predictors fails to converge. Check out the data `X_train` to see that this may be probably due to the predictors have different orders of magnitude. For example, the predictor `DiabetesPedigreeFunction` has values in [0.078, 2.42], while the predictor `Insulin` has values in [0, 800].

Let us fit the model to the scaled data.

```
# Create a model - not trained yet
logreg = LogisticRegression()

# Train the model
logreg.fit(X_train_scaled, y_train)
```

```
LogisticRegression()
```

The model converges to a solution with the scaled data!

The coefficients of the model can be returned with the `coef_` attribute of the `LogisticRegression()` object. However, the output is not as well formatted as in the case of the `statsmodels` library since `sklearn` is developed primarily for the purpose of prediction, and not inference.

```
# Use coef_ to return the coefficients - only log reg inference you can do with sklearn
print(logreg.coef_)
```

```
[[ 0.32572891  1.20110566 -0.32046591  0.06849882 -0.21727131  0.72619528
   0.40088897  0.29698818]]
```

1.4 Computing performance metrics

1.4.1 Accuracy

Let us test the model prediction accuracy on the test data. We'll demonstrate two different functions that can be used to compute model accuracy - `accuracy_score()`, and `score()`.

The `accuracy_score()` function from the `metrics` module of the `sklearn` library is general, and can be used for any classification model. We'll use it along with the `predict()` method of the `LogisticRegression()` object, which returns the predicted class based on a threshold probability of 0.5.

```
# Get the predicted classes first
y_pred = logreg.predict(X_test_scaled)

# Use the predicted and true classes for accuracy
print(accuracy_score(y_pred, y_test)*100)
```

```
73.37662337662337
```

The `score()` method of the `LogisticRegression()` object can be used to compute the accuracy only for a logistic regression model. Note that for a `LinearRegression()` object, the `score()` method will return the model R-squared.

```
# Use .score with test predictors and response to get the accuracy
# Implements the same thing under the hood
print(logreg.score(X_test_scaled, y_test)*100)
```

```
73.37662337662337
```

1.4.2 ROC-AUC

The `roc_curve()` and `auc()` functions from the `metrics` module of the `sklearn` library can be used to compute the ROC-AUC, or the area under the ROC curve. Note that for computing ROC-AUC, we need the predicted probability, instead of the predicted class. Thus, we'll use the `predict_proba()` method of the `LogisticRegression()` object, which returns the predicted probability for the observation to belong to each of the classes, instead of using the `predict()` method, which returns the predicted class based on threshold probability of 0.5.

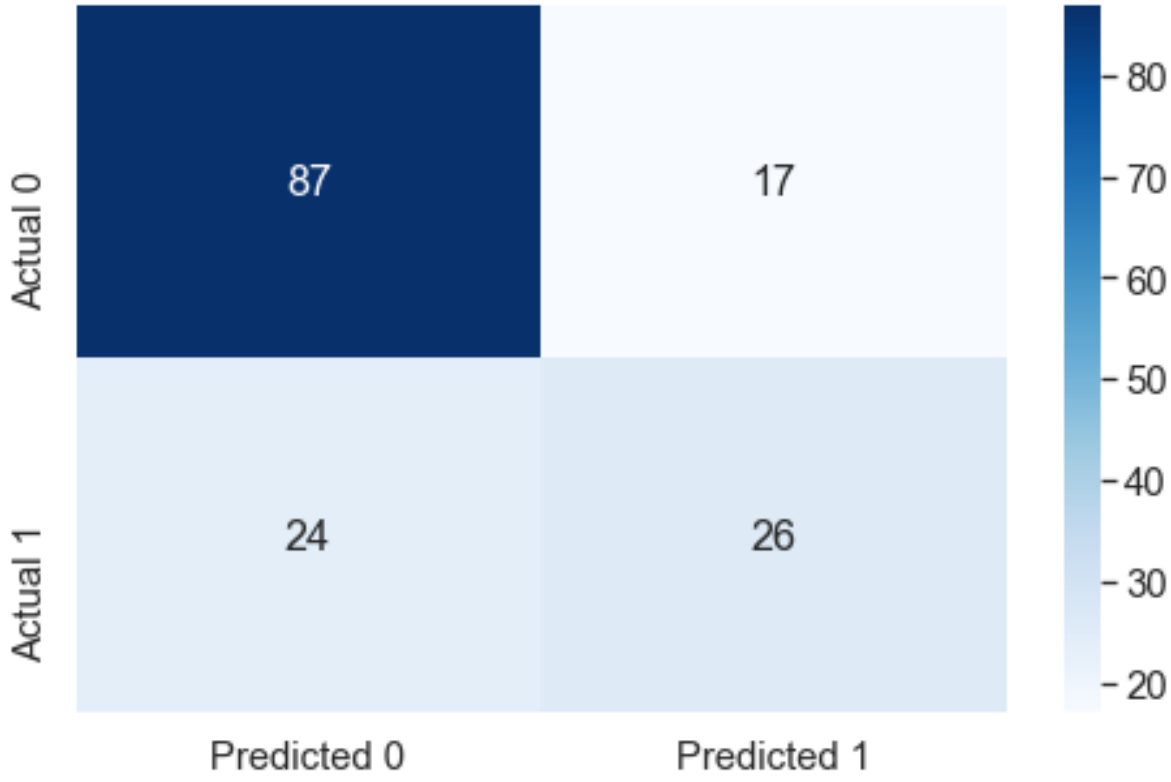
```
#Computing the predicted probability for the observation to belong to the positive class (
#The 2nd column in the output of predict_proba() consists of the probability of the observ
#belong to the positive class (y=1)
y_pred_prob = logreg.predict_proba(X_test_scaled)[:,1]

#Using the predicted probability computed above to find ROC-AUC
fpr, tpr, auc_thresholds = roc_curve(y_test, y_pred_prob)
print(auc(fpr, tpr))# AUC of ROC
```

```
0.7923076923076922
```

1.4.3 Confusion matrix & precision-recall

The `confusion_matrix()`, `precision_score()`, and `recall_score()` functions from the `metrics` module of the `sklearn` library can be used to compute the confusion matrix, precision, and recall respectively.

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




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

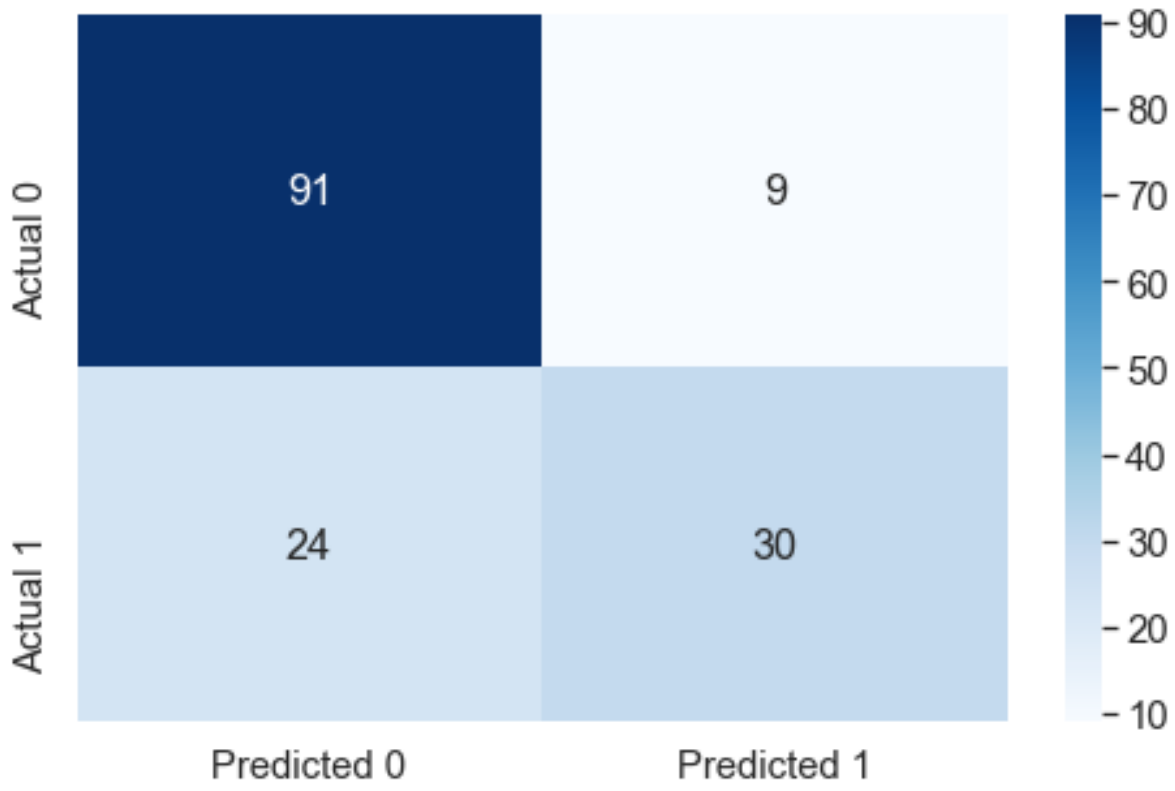
```
Precision:  0.6046511627906976
Recall:  0.52
```

Let us compute the performance metrics if we develop the model using stratified splitting.

```
# Developing the model with stratified splitting

#Scaling data
scaler = StandardScaler().fit(X_train_stratified)
X_train_stratified_scaled = scaler.transform(X_train_stratified)
X_test_stratified_scaled = scaler.transform(X_test_stratified)

# Training the model
logreg.fit(X_train_stratified_scaled, y_train_stratified)
```

```
#Computing the accuracy
y_pred_stratified = logreg.predict(X_test_stratified_scaled)
print("Accuracy: ",accuracy_score(y_pred_stratified, y_test_stratified)*100)

#Computing the ROC-AUC
y_pred_stratified_prob = logreg.predict_proba(X_test_stratified_scaled)[:,1]
fpr, tpr, auc_thresholds = roc_curve(y_test_stratified, y_pred_stratified_prob)
print("ROC-AUC: ",auc(fpr, tpr))# AUC of ROC

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

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

```
Accuracy:  78.57142857142857
ROC-AUC:  0.8505555555555556
Precision:  0.7692307692307693
Recall:  0.5555555555555556
```



The model with the stratified train-test split has a better performance as compared to the other model on all the performance metrics!

1.5 Tuning the model hyperparameters

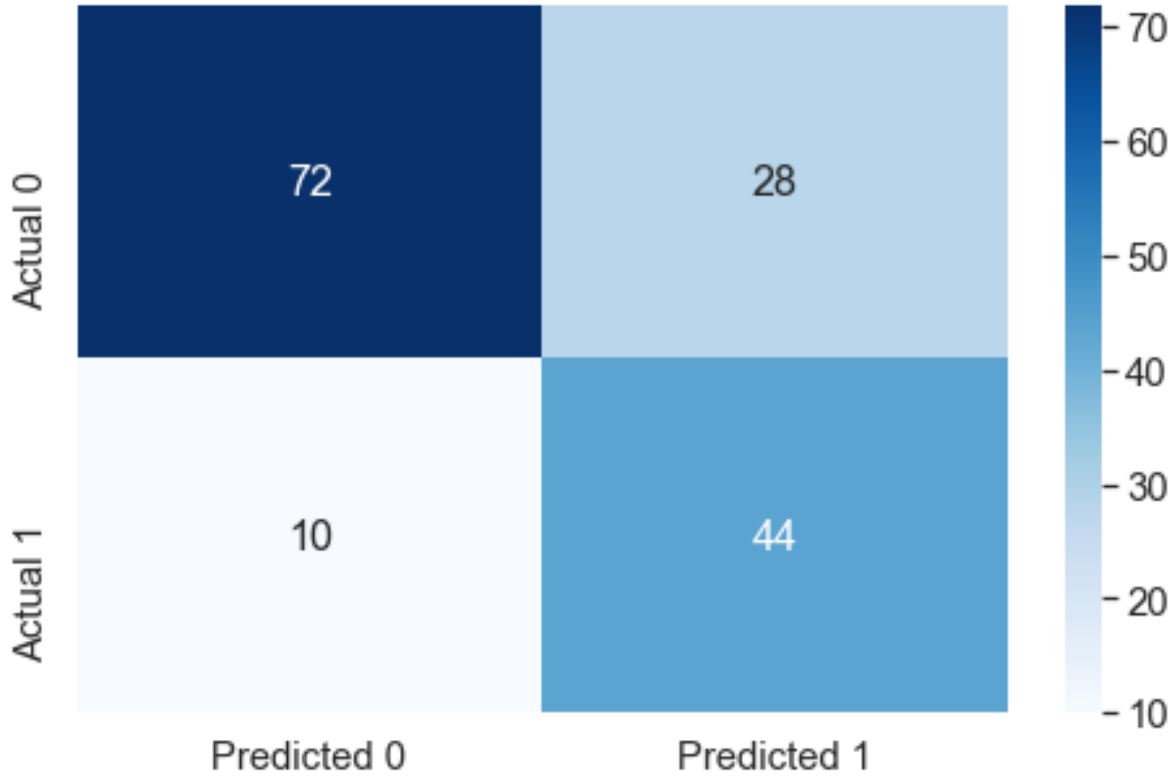
A hyperparameter (among others) that can be trained in a logistic regression model is the regularization parameter.

We may also wish to tune the decision threshold probability. Note that the decision threshold probability is not considered a hyperparameter of the model. Hyperparameters are model parameters that are set prior to training and cannot be directly adjusted by the model during training. Examples of hyperparameters in a logistic regression model include the regularization parameter, and the type of shrinkage penalty - lasso / ridge. These hyperparameters are typically optimized through a separate tuning process, such as cross-validation or grid search, before training the final model.

The performance metrics can be computed using a desired value of the threshold probability. Let us compute the performance metrics for a desired threshold probability of 0.3.

```
# Performance metrics computation for a desired threshold probability of 0.3
desired_threshold = 0.3

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

#Computing the accuracy
print("Accuracy: ",accuracy_score(y_pred_desired_threshold, y_test_stratified)*100)

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

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

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

```
Accuracy:  75.32467532467533
ROC-AUC:  0.8505555555555556
Precision:  0.6111111111111112
Recall:  0.8148148148148148
```



1.5.1 Tuning decision threshold probability

Suppose we wish to find the optimal decision threshold probability to maximize accuracy. Note that we cannot use the test dataset to optimize model hyperparameters, as that may lead to overfitting on the test data. We'll use K-fold cross validation on train data to find the optimal decision threshold probability.

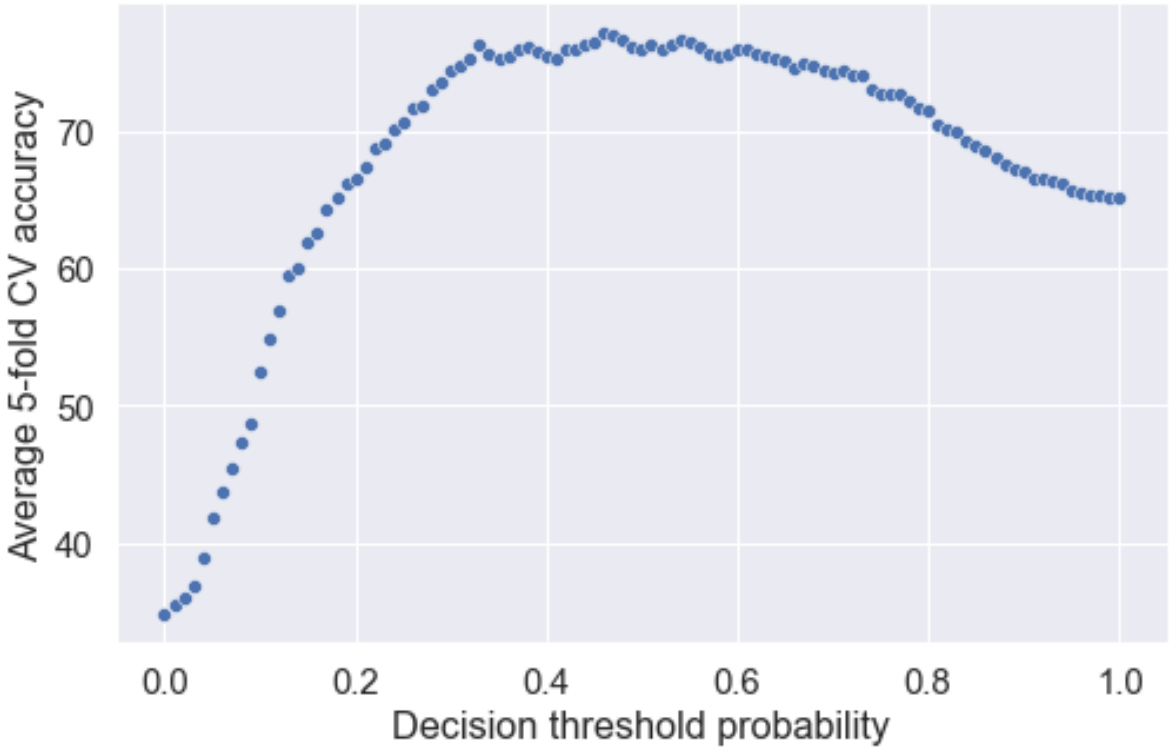
We'll use the `cross_val_predict()` function from the `model_selection` module of `sklearn` to compute the K-fold cross validated predicted probabilities. Note that this function simplifies the task of manually creating the K-folds, training the model K-times, and computing the predicted probabilities on each of the K-folds. Thereafter, the predicted probabilities will be used to find the optimal threshold probability that maximizes the classification accuracy.

```
hyperparam_vals = np.arange(0,1.01,0.01)
accuracy_iter = []

predicted_probability = cross_val_predict(LogisticRegression(), X_train_stratified_scaled,
                                                    y_train_stratified, cv = 5, method = 'predic
```

```
for threshold_prob in hyperparam_vals:
    predicted_class = predicted_probability[:,1] > threshold_prob
    predicted_class = predicted_class.astype(int)

    #Computing the accuracy
    accuracy = accuracy_score(predicted_class, y_train_stratified)*100
    accuracy_iter.append(accuracy)
```

Let us visualize the accuracy with change in decision threshold probability.

```
# Accuracy vs decision threshold probability
sns.scatterplot(x = hyperparam_vals, y = accuracy_iter)
plt.xlabel('Decision threshold probability')
plt.ylabel('Average 5-fold CV accuracy');
```



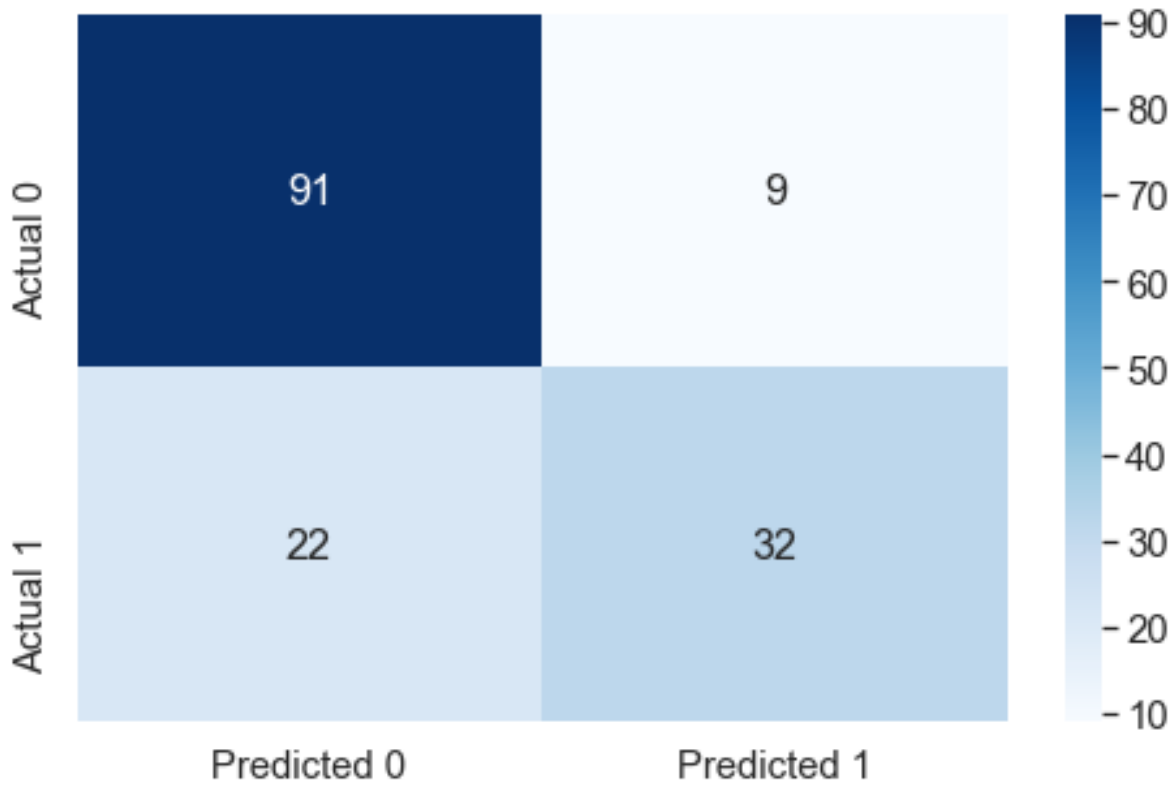
The optimal decision threshold probability is the one that maximizes the K-fold cross validation accuracy.

```
# Optimal decision threshold probability
hyperparam_vals[accuracy_iter.index(max(accuracy_iter))]
```

```
0.46
```

```
# Performance metrics computation for the optimum decision threshold probability
desired_threshold = 0.46

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

#Computing the accuracy
print("Accuracy: ",accuracy_score(y_pred_desired_threshold, y_test_stratified)*100)

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

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

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

```
Accuracy:  79.87012987012987
ROC-AUC:  0.8505555555555556
Precision:  0.7804878048780488
Recall:  0.5925925925925926
```



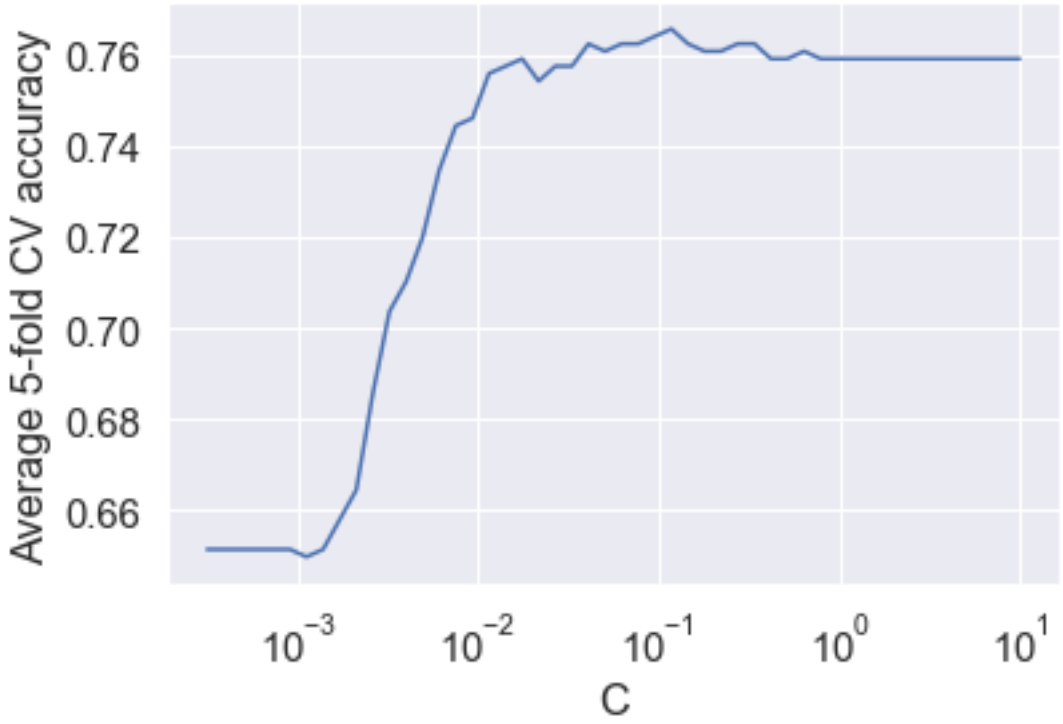
Model performance on test data has improved with the optimal decision threshold probability.

1.5.2 Tuning the regularization parameter

The `LogisticRegression()` method has a default *L2* regularization penalty, which means ridge regression.`C` is $1/\lambda$, where λ is the hyperparameter that is multiplied with the ridge penalty. `C` is 1 by default.

```
accuracy_iter = []
hyperparam_vals = 10**np.linspace(-3.5, 1)

for c_val in hyperparam_vals: # For each possible C value in your grid
    logreg_model = LogisticRegression(C=c_val) # Create a model with the C value

    accuracy_iter.append(cross_val_score(logreg_model, X_train_stratified_scaled, y_train_
                                         scoring='accuracy', cv=5)) # Find the cv results
```

```
plt.plot(hyperparam_vals, np.mean(np.array(accuracy_iter), axis=1))
plt.xlabel('C')
plt.ylabel('Average 5-fold CV accuracy')
plt.xscale('log')
plt.show()
```



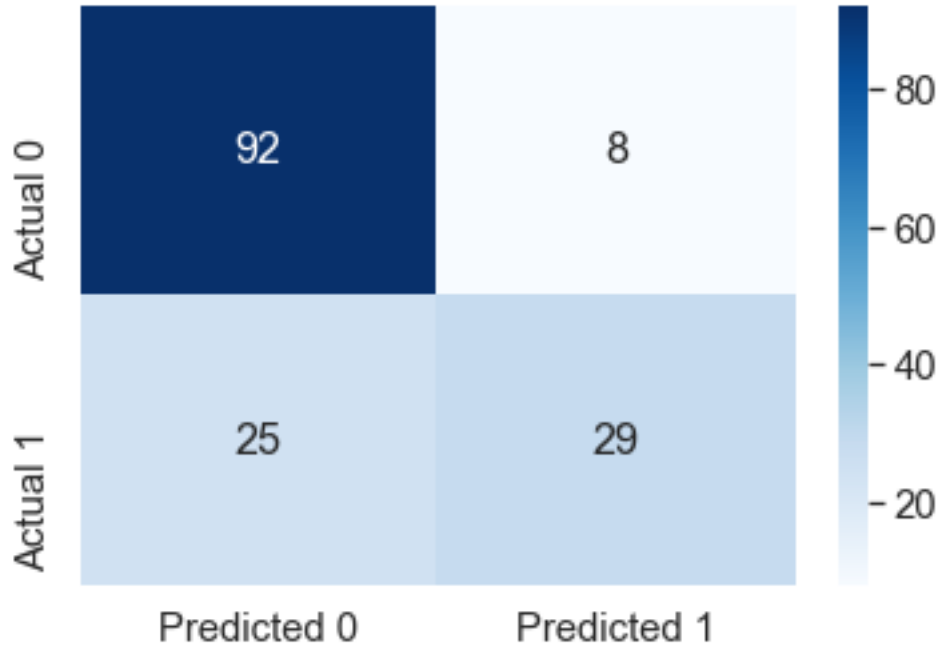

```
# Optimal value of the regularization parameter 'C'
optimal_C = hyperparam_vals[np.argmax(np.array(accuracy_iter).mean(axis=1))]
optimal_C
```

```
0.11787686347935879
```

```
# Developing the model with stratified splitting and optimal 'C'

#Scaling data
scaler = StandardScaler().fit(X_train_stratified)
X_train_stratified_scaled = scaler.transform(X_train_stratified)
X_test_stratified_scaled = scaler.transform(X_test_stratified)
```

```
# Training the model
logreg = LogisticRegression(C = optimal_C)
logreg.fit(X_train_stratified_scaled, y_train_stratified)

#Computing the accuracy
y_pred_stratified = logreg.predict(X_test_stratified_scaled)
print("Accuracy: ",accuracy_score(y_pred_stratified, y_test_stratified)*100)

#Computing the ROC-AUC
y_pred_stratified_prob = logreg.predict_proba(X_test_stratified_scaled)[:,1]
fpr, tpr, auc_thresholds = roc_curve(y_test_stratified, y_pred_stratified_prob)
print("ROC-AUC: ",auc(fpr, tpr))# AUC of ROC

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

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

```
Accuracy:  78.57142857142857
ROC-AUC:  0.8516666666666666
Precision:  0.7837837837837838
Recall:  0.5370370370370371
```



1.5.3 Tuning the decision threshold probability and the regularization parameter simultaneously

```
threshold_hyperparam_vals = np.arange(0,1.01,0.01)
C_hyperparam_vals = 10**np.linspace(-3.5, 1)
accuracy_iter = pd.DataFrame(columns = {'threshold', 'C', 'accuracy'})
iter_number = 0

for c_val in C_hyperparam_vals:
    predicted_probability = cross_val_predict(LogisticRegression(C = c_val), X_train_strat
                                              y_train_stratified, cv = 5, method = 'pr

    for threshold_prob in threshold_hyperparam_vals:
        predicted_class = predicted_probability[:,1] > threshold_prob
        predicted_class = predicted_class.astype(int)

        #Computing the accuracy
        accuracy = accuracy_score(predicted_class, y_train_stratified)*100
        accuracy_iter.loc[iter_number, 'threshold'] = threshold_prob
        accuracy_iter.loc[iter_number, 'C'] = c_val
        accuracy_iter.loc[iter_number, 'accuracy'] = accuracy
```

```
        iter_number = iter_number + 1
```

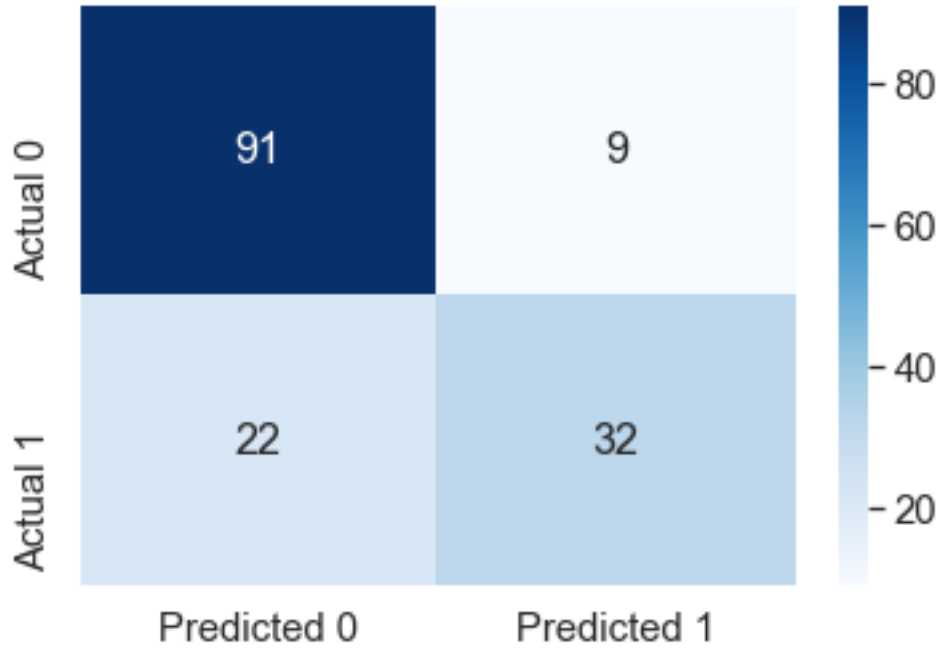
```
# Parameters for highest accuracy
optimal_C = accuracy_iter.sort_values(by = 'accuracy', ascending = False).iloc[0,:]['C']
optimal_threshold = accuracy_iter.sort_values(by = 'accuracy', ascending = False).iloc[0,

#Optimal decision threshold probability
print("Optimal decision threshold = ", optimal_threshold)

#Optimal C
print("Optimal C = ", optimal_C)
```

```
Optimal decision threshold =  0.46
Optimal C =  4.291934260128778
```

```
# Developing the model with stratified splitting, optimal decision threshold probability,

#Scaling data
scaler = StandardScaler().fit(X_train_stratified)
X_train_stratified_scaled = scaler.transform(X_train_stratified)
X_test_stratified_scaled = scaler.transform(X_test_stratified)

# Training the model
logreg = LogisticRegression(C = optimal_C)
logreg.fit(X_train_stratified_scaled, y_train_stratified)

# Performance metrics computation for the optimal threshold probability
y_pred_stratified_prob = logreg.predict_proba(X_test_stratified_scaled)[:,1]

# Classifying observations in the positive class (y = 1) if the predicted probability is g
# than the desired decision threshold probability
y_pred_desired_threshold = y_pred_stratified_prob > optimal_threshold
y_pred_desired_threshold = y_pred_desired_threshold.astype(int)

#Computing the accuracy
print("Accuracy: ",accuracy_score(y_pred_desired_threshold, y_test_stratified)*100)

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

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

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

```
Accuracy:  79.87012987012987
ROC-AUC:  0.8509259259259259
Precision:  0.7804878048780488
Recall:  0.5925925925925926
```



Later in the course, we'll see the `sklearn` function `GridSearchCV`, which is used to optimize several model hyperparameters simultaneously with K-fold cross validation, while avoiding `for` loops.

2 Regression splines

Read sections 7.1-7.4 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 statsmodels.formula.api as smf
import statsmodels.api as sm
import seaborn as sns
import matplotlib.pyplot as plt
from patsy import dmatrix
from sklearn.metrics import mean_squared_error
# install the py-earth library first: https://anaconda.org/conda-forge/sklearn-contrib-py-
from pyearth import Earth
from sklearn.linear_model import LinearRegression
```

```
#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

2.1 Polynomial regression vs Regression splines

2.1.1 Model of degree 1

```
X = pd.DataFrame(train['mileage'])
X_test = pd.DataFrame(test['mileage'])
y = train['price']
lr_model = LinearRegression()
lr_model.fit(X, y);
```

```
#Regression spline of degree 1

#Creating basis functions for splines of degree 1
transformed_x = dmatrix("bs(mileage , knots=(33000,66000,100000), degree = 1, include_inte
                        data = trainf,return_type = 'dataframe')
```

```
transformed_x.head()
```

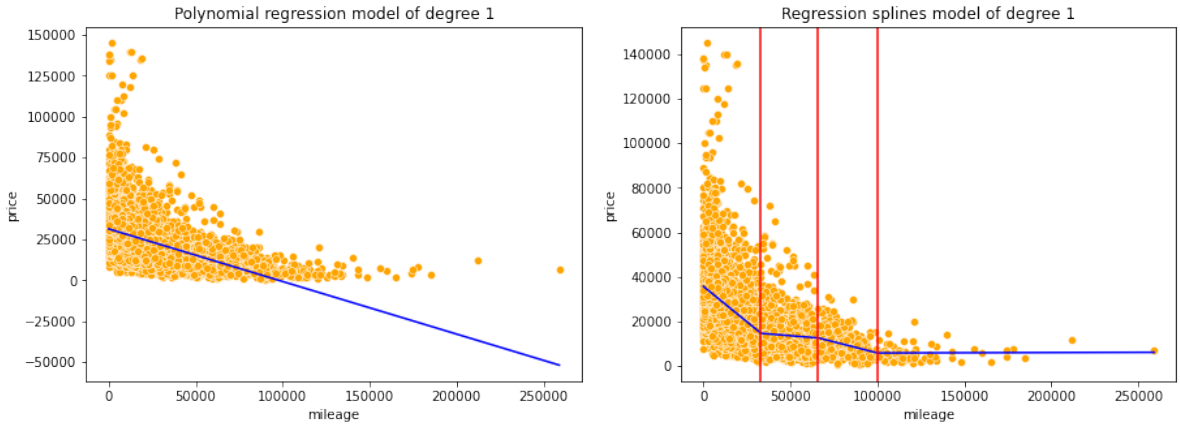
	Intercept	bs(mileage, knots=(33000, 66000, 100000), degree=1, include_intercept=False)[0]	bs(mileage,
0	1.0	0.000303	0.000000
1	1.0	0.327646	0.000000
2	1.0	0.000152	0.000000
3	1.0	0.572563	0.000000
4	1.0	0.092333	0.907667

Note that the truncated power basis in the class presentation is conceptually simple to understand, it may run into numerical issues as powers of large numbers can lead to severe rounding errors. The `bs()` function generates the B-spline basis, which allows for efficient computation, especially in case of a large number of knots. All the basis function values are normalized to be in [0, 1] in the B-spline basis. Although we'll use the B-spline basis functions to fit splines, details regarding the B-spline basis functions are not included in the syllabus.

We actually don't need to separately generate basis functions, and then fit the model. We can do it in the same line of code using the `statsmodels OLS` method.

```
# Regression spline model with linear splines
reg_spline_model = smf.ols('price~bs(mileage, knots = (33000,66000,100000), degree = 1, in
                           data = train).fit()
```

```
#Visualizing polynomial model and the regression spline model of degree 1

knots = [33000,66000,100000] #Knots for the spline
d=1 #Degree of predictor in the model
#Writing a function to visualize polynomial model and the regression spline model of degre
def viz_models():
    fig, axes = plt.subplots(1,2,figsize = (15,5))
    plt.subplots_adjust(wspace=0.2)

    #Visualizing the linear regression model
    pred_price = lr_model.predict(X)
    sns.scatterplot(ax = axes[0],x = 'mileage', y = 'price', data = train, color = 'orange
    sns.lineplot(ax = axes[0],x = train.mileage, y = pred_price, color = 'blue')
    axes[0].set_title('Polynomial regression model of degree '+str(d))

    #Visualizing the regression splines model of degree 'd'
    axes[1].set_title('Regression splines model of degree '+ str(d))
    sns.scatterplot(ax=axes[1],x = 'mileage', y = 'price', data = train, color = 'orange')
    sns.lineplot(ax=axes[1],x = train.mileage, y = reg_spline_model.predict(), color = 'bl
    for i in range(3):
        plt.axvline(knots[i], 0,100,color='red')
viz_models()
```



We observe the regression splines model better fits the data as compared to the polynomial regression model. This is because regression splines of degree 1 fit piecewise polynomials, or linear models on sub-sections of the predictor, which helps better capture the trend. However, this added flexibility may also lead to overfitting. Hence, one must be careful to check for overfitting when using splines. Overfitting may be checked by k-fold cross validation or comparing

test and train errors.

The red lines in the plot on the right denote the position of knots. Knots separate distinct splines.

Although, we can separately generate the basis functions for test data, it may lead to incaccurate results if the distribution of the predictor values in test data is different from that in the train data. This is because the B-spline basis functions of train data are generated after normalizing the predictor values. If the basis functions of test data are generated independently, their values may be inaccurate, as they will depend on the domain space spanned by the test data.

```
# Basis functions for test data - avoid generating basis functions separately for test dat
# as the test data normailization may be different from the train data normalization
test_x = dmatrix("bs(mileage , knots=(33000,66000,100000), degree = 1, include_intercept =
```

```
#Function to compute RMSE (root mean squared error on train and test datasets)
def rmse():
    #Error on train data for the linear regression model
    print("RMSE on train data:")
    print("Linear regression:", np.sqrt(mean_squared_error(lr_model.predict(X),train.price

    #Error on train data for the regression spline model
    print("Regression splines:", np.sqrt(mean_squared_error(reg_spline_model.predict(X),tr

    #Error on test data for the linear regression model
    print("\nRMSE on test data:")
    print("Linear regression:",np.sqrt(mean_squared_error(lr_model.predict(X_test),test.pr

    #Error on test data for the regression spline model
    print("Regression splines:",np.sqrt(mean_squared_error(reg_spline_model.predict(X_test
rmse()
```

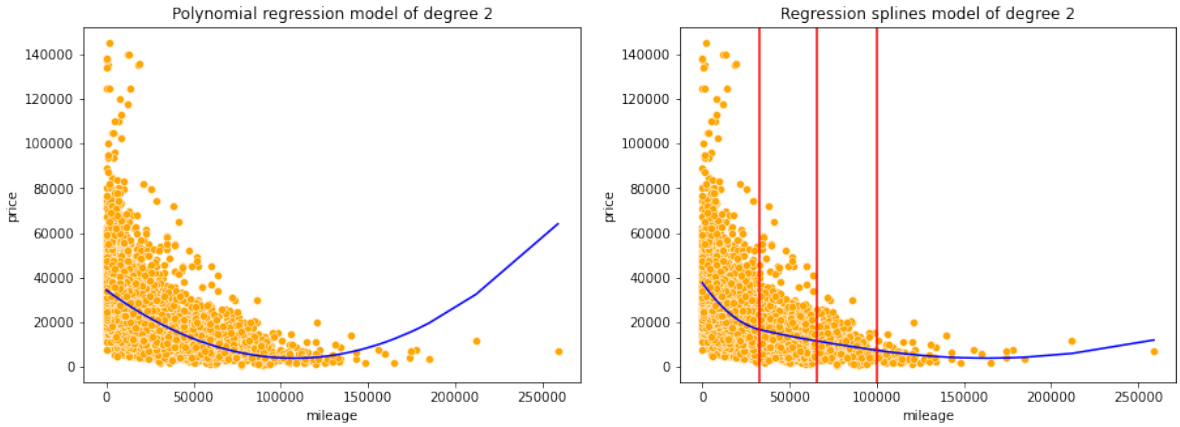
```
RMSE on train data:
Linear regression: 14403.250083261853
Regression splines: 13859.640716531134

RMSE on test data:
Linear regression: 14370.94086395544
Regression splines: 13770.118474361932
```

2.1.2 Model of degree 2

A higher degree model will lead to additional flexibility for both polynomial and regression splines models.

```
#Including mileage squared as a predictor and developing the model
ols_object = smf.ols(formula = 'price~mileage+I(mileage**2)', data = train)
lr_model = ols_object.fit()

#Regression spline of degree 2
reg_spline_model = smf.ols('price~bs(mileage, knots = (33000,66000,100000), degree = 2, in
                           data = train).fit()
```

```
d=2
viz_models()
```



Unlike polynomial regression, splines functions avoid imposing a global structure on the non-linear function of X. This provides a better local fit to the data.

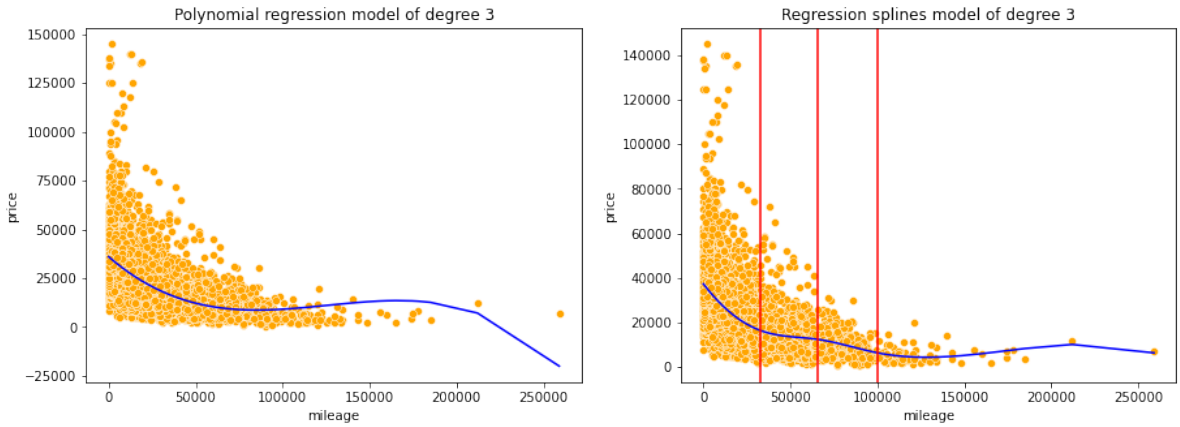
```
rmse()
```

```
RMSE on train data:
Linear regression: 14009.819556665143
Regression splines: 13818.572654146721

RMSE on test data:
Linear regression: 13944.20691909441
Regression splines: 13660.777953039395
```

2.1.3 Model of degree 3

```
#Including mileage cube squared as a predictor and developing the model
ols_object = smf.ols(formula = 'price~mileage+I(mileage**2)+I(mileage**3)', data = train)
lr_model = ols_object.fit()

#Regression spline of degree 3
reg_spline_model = smf.ols('price~bs(mileage, knots = (33000,66000,100000), degree = 3, in
                           data = train).fit()
```

```
d=3
viz_models()
```



Unlike polynomial regression, splines functions avoid imposing a global structure on the non-linear function of X. This provides a better local fit to the data.

```
rmse()
```

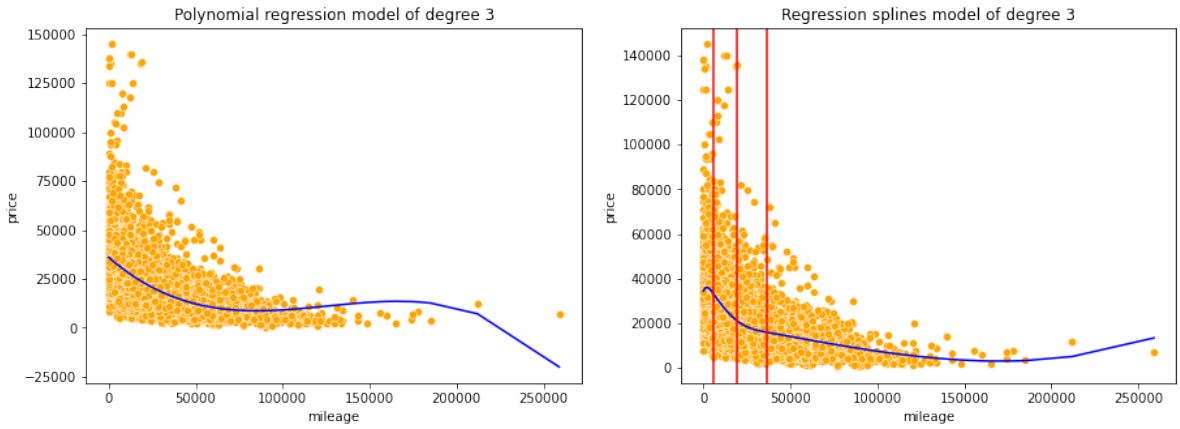
```
RMSE on train data:
Linear regression: 13891.962447594644
Regression splines: 13822.70511947823

RMSE on test data:
Linear regression: 13789.708418357186
Regression splines: 13683.776494331632
```

2.2 Regression splines with knots at uniform quantiles of data

If degrees of freedom are provided instead of knots, the knots are by default chosen at uniform quantiles of data. For example if there are 7 degrees of freedom (including the intercept), then there will be 7-4 = 3 knots. These knots will be chosen at the 25th, 50th and 75th quantiles of the data.

```
#Regression spline of degree 3

#Regression spline of degree 3 with knots at uniform quantiles of data
reg_spline_model = smf.ols('price~bs(mileage, df = 6, degree = 3, include_intercept = Fals
                           data = train).fit()
```

```
d=3
unif_knots = pd.qcut(train.mileage,4,retbins=True)[1][1:4]
knots=unif_knots
viz_models()
```



Splines can be unstable at the outer range of predictors. Note that splines are themselves piecewise polynomials with no constraints at the 2 extreme ends of the predictor space. Thus, they may become unstable at those 2 ends. In the right scatter plot, we can see that price has a decreasing trend with mileage. However on the extreme left of the plot, we see the trend reversing with regard to the model, which suggests potential overfitting. Also, from the domain knowledge about cars we know that there is no reason why price will reduce if the car is relatively new. Thus, there may be overfitting with cubic splines at / near the extreme points of the domain space. In the figure (on the right), the left-most spline may be overfitting.

This motivates us to introduce natural cubic splines (below), which help with the stability at extreme points by enforcing the spline to be linear at those points. We may also think about

it as another kind of a "knot" being put at the two ends to make the spline stable at these points.

```
rmse()
```

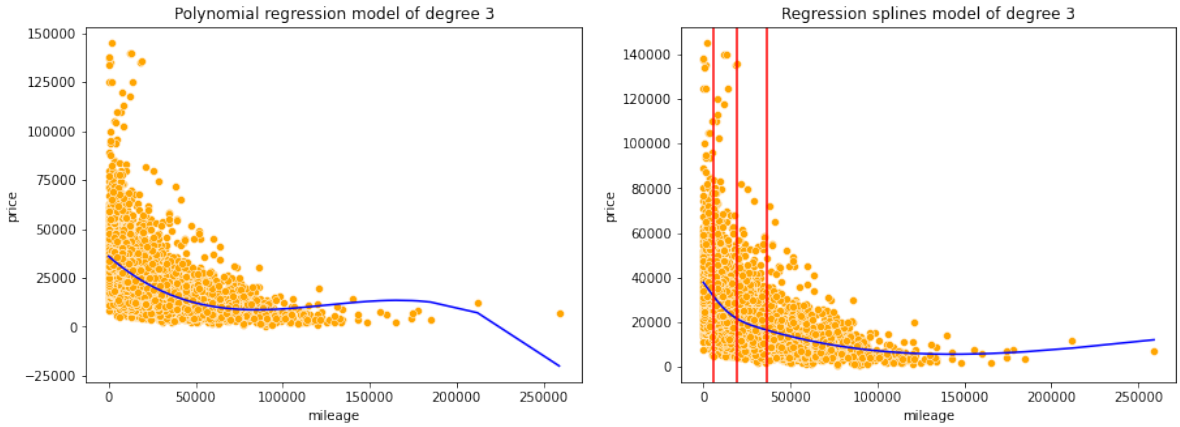
```
RMSE on train data:
Linear regression: 13891.962447594644
Regression splines: 13781.79102252679

RMSE on test data:
Linear regression: 13789.708418357186
Regression splines: 13605.726076704668
```

2.3 Natural cubic splines

Page 298: "A natural spline is a regression spline with additional boundary constraints: the function is required to be linear at the boundary (in the region where X is smaller than the smallest knot, or larger than the largest knot). This additional constraint means that natural splines generally produce more stable estimates at the boundaries."

```
#Natural cubic spline

#Creating basis functions for the natural cubic spline
reg_spline_model = smf.ols('price~cr(mileage, df = 4)',
                           data = train).fit()
```

```
d=3;
unif_knots = pd.qcut(train.mileage,4,retbins=True)[1][1:4]
knots=unif_knots
viz_models()
```



Note that the natural cubic spline is more stable than a cubic splines with knots at uniformly distributed quantiles.

```
rmse()
```

```
RMSE on train data:
Linear regression: 13891.962447594644
Regression splines: 13826.125469174143

RMSE on test data:
Linear regression: 13789.708418357186
Regression splines: 13660.35327661836
```

2.4 Generalized additive model (GAM)

GAM allows for flexible nonlinearities in several variables, but retain the additive structure of linear models. In a GAM, non-linear basis functions of predictors can be used as predictors of a linear regression model. For example,

$$y = f_1(X_1) + f_2(X_2) + \epsilon$$

is a GAM, where $f_1(.)$ may be a cubic spline based on the predictor X_1, and $f_2(.)$ may be a step function based on the predictor X_2.

```
#GAM
#GAM includes cubic splines for mileage. Other predictors are year, engineSize, mpg, milea
model_gam = smf.ols('price~bs(mileage,df=6,degree = 3)+year*engineSize*mpg*mileage', data
```

```
preds = model_gam.predict(test)
np.sqrt(mean_squared_error(preds,test.price))
```

```
8393.773177637542
```

```
#GAM
#GAM includes cubic splines for mileage, year, engineSize, mpg, and interactions of all pr
model_gam = smf.ols('price~bs(mileage,df=6,degree = 3)+bs(mpg,df=6,degree = 3)+\
bs(engineSize,df=6,degree = 3)+year*engineSize*mpg*mileage', data = train).fit()

preds = model_gam.predict(test)
np.sqrt(mean_squared_error(preds,test.price))
```

```
7981.100853841914
```

```
ols_object = smf.ols(formula = 'price~(year+engineSize+mileage+mpg)**2+I(mileage**2)+I(mil
model = ols_object.fit()
model.summary()
```

Table 2.3: OLS Regression Results

Dep. Variable:	price	R-squared:	0.704
Model:	OLS	Adj. R-squared:	0.703
Method:	Least Squares	F-statistic:	1308.
Date:	Sun, 09 Apr 2023	Prob (F-statistic):	0.00
Time:	20:48:35	Log-Likelihood:	-52157.
No. Observations:	4960	AIC:	1.043e+05
Df Residuals:	4950	BIC:	1.044e+05
Df Model:	9		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[0.025	0.975]
Intercept	-0.0009	0.000	-2.752	0.006	-0.002	-0.000
year	-1.1470	0.664	-1.728	0.084	-2.448	0.154
engineSize	0.0052	0.000	17.419	0.000	0.005	0.006
mileage	-31.4751	2.621	-12.010	0.000	-36.613	-26.337
mpg	-0.0201	0.002	-13.019	0.000	-0.023	-0.017
year:engineSize	9.5957	0.254	37.790	0.000	9.098	10.094

year:mileage	0.0154	0.001	11.816	0.000	0.013	0.018
year:mpg	0.0572	0.013	4.348	0.000	0.031	0.083
engineSize:mileage	-0.1453	0.008	-18.070	0.000	-0.161	-0.130
engineSize:mpg	-98.9062	11.832	-8.359	0.000	-122.102	-75.710
mileage:mpg	0.0011	0.000	2.432	0.015	0.000	0.002
I(mileage ** 2)	7.713e-06	3.75e-07	20.586	0.000	6.98e-06	8.45e-06
I(mileage ** 3)	-1.867e-11	1.43e-12	-13.077	0.000	-2.15e-11	-1.59e-11

Omnibus:	1830.457	Durbin-Watson:	0.634
Prob(Omnibus):	0.000	Jarque-Bera (JB):	34927.811
Skew:	1.276	Prob(JB):	0.00
Kurtosis:	15.747	Cond. No.	2.50e+18

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

```
9026.775740000594
```

Note the RMSE with GAM that includes regression splines for mileage is lesser than that of the linear regression model, indicating a better fit.

2.5 MARS (Multivariate Adaptive Regression Splines)

```
from pyearth import Earth
X=pd.DataFrame(train['mileage'])
y=pd.DataFrame(train['price'])
```

2.5.1 MARS of degree 1

```
model = Earth(max_terms=500, max_degree=1) # note, terms in brackets are the hyperparamete
model.fit(X,y)
```

```
C:\Users\akl0407\Anaconda3\lib\site-packages\pyearth\earth.py:813: FutureWarning: `rcond` pa
To use the future default and silence this warning we advise to pass `rcond=None`, to keep us
  pruning_passer.run()
```

```
C:\Users\akl0407\Anaconda3\lib\site-packages\pyearth\earth.py:1066: FutureWarning: `rcond` pa
To use the future default and silence this warning we advise to pass `rcond=None`, to keep us
  coef, resid = np.linalg.lstsq(B, weighted_y[:, i])[0:2]
```

```
Earth(max_degree=1, max_terms=500)
```

```
print(model.summary())
```

```
Earth Model
-------------------------------------
Basis Function  Pruned  Coefficient
-------------------------------------
(Intercept)     No      -553155
h(x0-22141)     Yes     None
h(22141-x0)     Yes     None
h(x0-3354)      No      -6.23571
h(3354-x0)      Yes     None
h(x0-15413)     No      -36.9613
h(15413-x0)     No      38.167
h(x0-106800)    Yes     None
h(106800-x0)    No      0.221844
h(x0-500)       No      170.039
h(500-x0)       Yes     None
h(x0-741)       Yes     None
h(741-x0)       No      -54.5265
h(x0-375)       No      -126.804
h(375-x0)       Yes     None
h(x0-2456)      Yes     None
h(2456-x0)      No      7.04609
-------------------------------------
MSE: 188429705.7549, GCV: 190035470.5664, RSQ: 0.2998, GRSQ: 0.2942
```

Model equation:

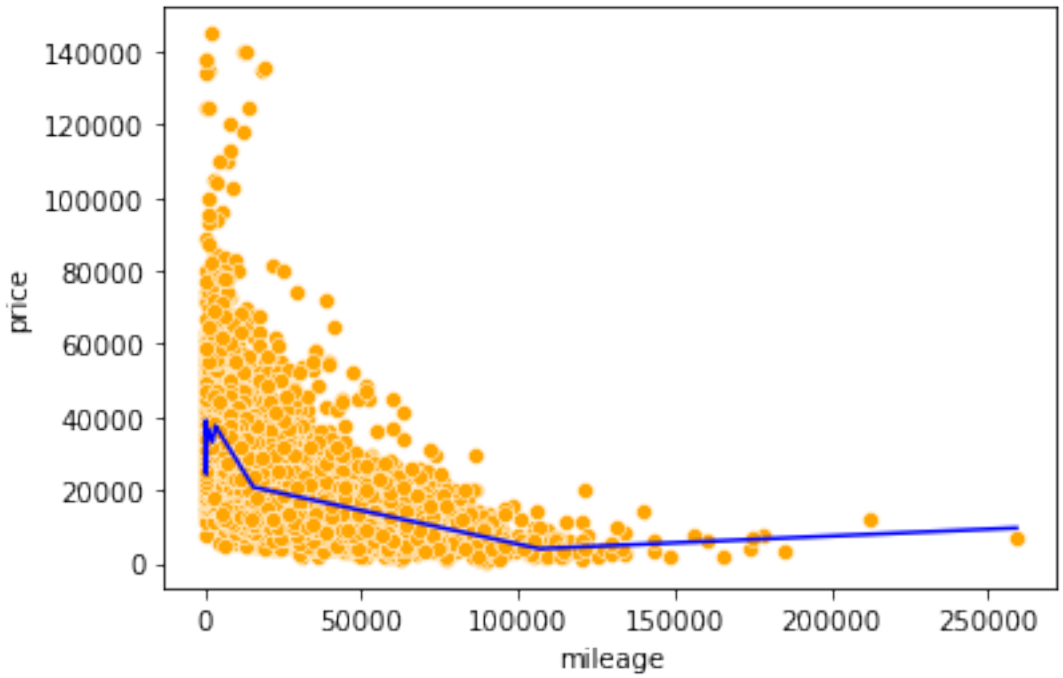
$$-553155 - 6.23(h(x0 - 3354)) - 36.96(h(x0 - 15413) + ... - 7.04(h(2456 - x0)$$

```
pred = model.predict(test.mileage)
np.sqrt(mean_squared_error(pred,test.price))
```

```
13650.2113154515
```

```
sns.scatterplot(x = 'mileage', y = 'price', data = train, color = 'orange')
sns.lineplot(x = train.mileage, y = model.predict(train.mileage), color = 'blue')
```

```
<AxesSubplot:xlabel='mileage', ylabel='price'>
```



2.5.2 MARS of degree 2

```
model = Earth(max_terms=500, max_degree=2) # note, terms in brackets are the hyperparamete
model.fit(X,y)
print(model.summary())
```

```
Earth Model
----------------------------------------------
Basis Function           Pruned  Coefficient
----------------------------------------------
(Intercept)              No      19369.7
h(x0-22141)              Yes     None
h(22141-x0)              Yes     None
```

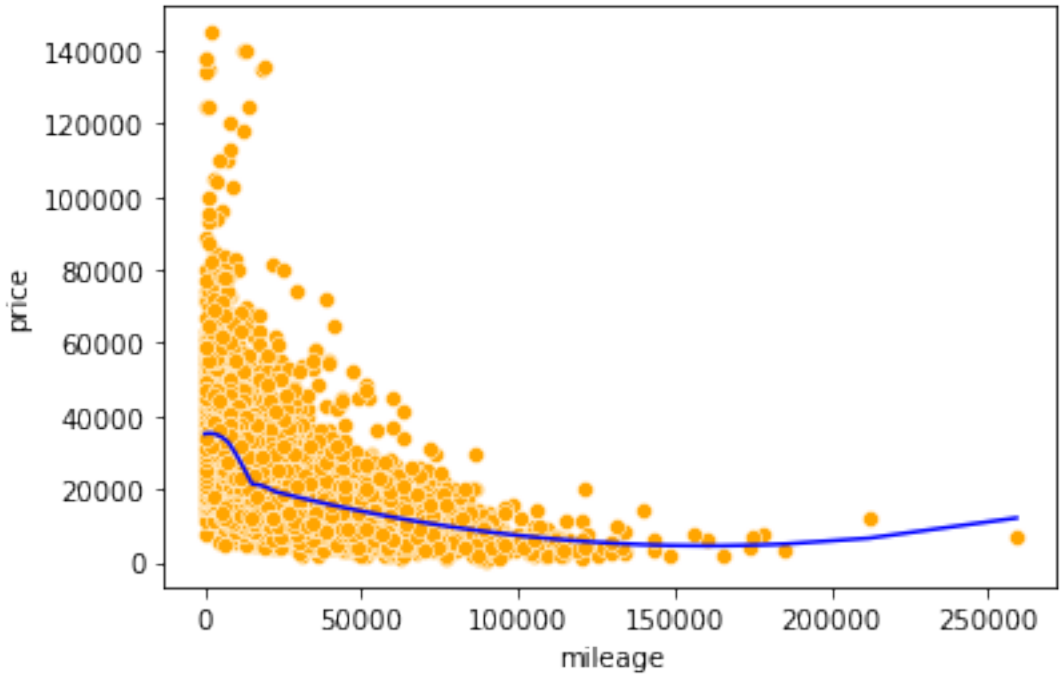
```
h(x0-7531)*h(22141-x0)   No      3.74934e-05
h(7531-x0)*h(22141-x0)   No      -6.74252e-05
x0*h(x0-22141)           No      -8.0703e-06
h(x0-15012)              Yes     None
h(15012-x0)              No      1.79813
h(x0-26311)*h(x0-22141)  No      8.85097e-06
h(26311-x0)*h(x0-22141)  Yes     None
-----------------------------------------------
MSE: 189264421.5682, GCV: 190298913.1652, RSQ: 0.2967, GRSQ: 0.2932

C:\Users\akl0407\Anaconda3\lib\site-packages\pyearth\earth.py:813: FutureWarning: `rcond` par
To use the future default and silence this warning we advise to pass `rcond=None`, to keep us
  pruning_passer.run()
C:\Users\akl0407\Anaconda3\lib\site-packages\pyearth\earth.py:1066: FutureWarning: `rcond` pa
To use the future default and silence this warning we advise to pass `rcond=None`, to keep us
  coef, resid = np.linalg.lstsq(B, weighted_y[:, i])[0:2]
```

```
pred = model.predict(test.mileage)
np.sqrt(mean_squared_error(pred,test.price))
```

```
13590.995419204985
```

```
sns.scatterplot(x = 'mileage', y = 'price', data = train, color = 'orange')
sns.lineplot(x = train.mileage, y = model.predict(train.mileage), color = 'blue')
```

```
<AxesSubplot:xlabel='mileage', ylabel='price'>
```



MARS provides a better fit than the splines that we used above. This is because MARS tunes the positions of the knots and considers interactions (also with tuned knots) to improve the model fit. Tuning of knots may improve the fit of splines as well.

2.5.3 MARS including categorical variables

```
#A categorical variable can be turned to dummy variables to use the Earth package for fitt
train_cat = pd.get_dummies(train)
test_cat = pd.get_dummies(test)
```

```
train_cat.head()
```

	carID	year	mileage	tax	mpg	engineSize	price	brand_audi	brand_bmw	brand_ford	...
0	18473	2020	11	145	53.3282	3.0	37980	0	1	0	...
1	15064	2019	10813	145	53.0430	3.0	33980	0	1	0	...
2	18268	2020	6	145	53.4379	3.0	36850	0	1	0	...
3	18480	2017	18895	145	51.5140	3.0	25998	0	1	0	...
4	18492	2015	62953	160	51.4903	3.0	18990	0	1	0	...

```
X = train_cat[['mileage','mpg','engineSize','year','fuelType_Diesel','fuelType_Electric',
               'fuelType_Hybrid','fuelType_Petrol']]
Xtest = test_cat[['mileage','mpg','engineSize','year','fuelType_Diesel','fuelType_Electric
                  'fuelType_Hybrid','fuelType_Petrol']]
```

```
model = Earth(max_terms=500, max_degree=2) # note, terms in brackets are the hyperparamete
model.fit(X,y)
print(model.summary())
```

```
C:\Users\akl0407\Anaconda3\lib\site-packages\pyearth\earth.py:813: FutureWarning: `rcond` pa
To use the future default and silence this warning we advise to pass `rcond=None`, to keep us
  pruning_passer.run()

Earth Model
------------------------------------------------------------
Basis Function                      Pruned  Coefficient
------------------------------------------------------------
(Intercept)                         No      2.17604e+06
h(engineSize-5.5)                   No      9.80752e+06
h(5.5-engineSize)                   No      1.92817e+06
h(mileage-21050)                    No      18.687
h(21050-mileage)                    No      -177.871
h(mileage-21050)*h(5.5-engineSize)  Yes     None
h(21050-mileage)*h(5.5-engineSize)  No      -0.224909
year                                No      4126.41
h(mpg-53.3495)                      No      344595
h(53.3495-mpg)                      Yes     None
fuelType_Hybrid*h(5.5-engineSize)   No      6124.34
h(mileage-21050)*year               No      -0.00930239
h(21050-mileage)*year               No      0.0886455
h(engineSize-5.5)*year              No      -4864.84
h(5.5-engineSize)*year              No      -952.92
h(mileage-1422)*h(53.3495-mpg)      No      -16.62
h(1422-mileage)*h(53.3495-mpg)      No      16.4306
fuelType_Hybrid                     No      -89090.6
h(mpg-21.1063)*h(53.3495-mpg)       Yes     None
h(21.1063-mpg)*h(53.3495-mpg)       No      -8815.99
h(mpg-23.4808)*h(5.5-engineSize)    No      -3649.97
h(23.4808-mpg)*h(5.5-engineSize)    Yes     None
h(mpg-20.5188)*year                 No      31.7341
h(20.5188-mpg)*year                 Yes     None
```

```
h(mpg-22.2566)*h(53.3495-mpg)       No      -52.2531
h(22.2566-mpg)*h(53.3495-mpg)       No      7916.19
h(mpg-22.6767)                      No      7.56432e+06
h(22.6767-mpg)                      Yes     None
h(mpg-23.9595)*h(mpg-22.6767)       Yes     None
h(23.9595-mpg)*h(mpg-22.6767)       No      -63225.4
h(mpg-21.4904)*h(22.6767-mpg)       No      -149055
h(21.4904-mpg)*h(22.6767-mpg)       Yes     None
h(mpg-21.1063)                      No      -887098
h(21.1063-mpg)                      Yes     None
h(mpg-29.5303)*h(mpg-22.6767)       No      -3028.87
h(29.5303-mpg)*h(mpg-22.6767)       Yes     None
h(mpg-28.0681)*h(5.5-engineSize)    No      3572.89
h(28.0681-mpg)*h(5.5-engineSize)    Yes     None
engineSize*h(5.5-engineSize)        No      -2952.65
h(mpg-25.3175)*h(mpg-21.1063)       No      -332551
h(25.3175-mpg)*h(mpg-21.1063)       No      324298
fuelType_Petrol*year                No      -1.37031
h(mpg-68.9279)*fuelType_Hybrid      No      -4087.9
h(68.9279-mpg)*fuelType_Hybrid      Yes     None
h(mpg-31.5043)*h(5.5-engineSize)    Yes     None
h(31.5043-mpg)*h(5.5-engineSize)    No      3691.82
h(mpg-32.7011)*h(5.5-engineSize)    Yes     None
h(32.7011-mpg)*h(5.5-engineSize)    No      -2262.78
h(mpg-44.9122)*h(mpg-22.6767)       No      335577
h(44.9122-mpg)*h(mpg-22.6767)       No      -335623
h(engineSize-5.5)*h(mpg-21.1063)    No      27815
h(5.5-engineSize)*h(mpg-21.1063)    Yes     None
h(mpg-78.1907)*fuelType_Hybrid      Yes     None
h(78.1907-mpg)*fuelType_Hybrid      No      2221.49
h(mpg-63.1632)*h(mpg-22.6767)       Yes     None
h(63.1632-mpg)*h(mpg-22.6767)       No      21.0093
fuelType_Hybrid*h(mpg-53.3495)      No      4121.91
h(mileage-22058)*h(53.3495-mpg)     No      16.6177
h(22058-mileage)*h(53.3495-mpg)     No      -16.6044
h(mpg-21.8985)                      Yes     None
h(21.8985-mpg)                      No      371659
---------------------------------------------------------
MSE: 45859836.5623, GCV: 47884649.3622, RSQ: 0.8296, GRSQ: 0.8221

C:\Users\akl0407\Anaconda3\lib\site-packages\pyearth\earth.py:1066: FutureWarning: `rcond` pa
To use the future default and silence this warning we advise to pass `rcond=None`, to keep us
  coef, resid = np.linalg.lstsq(B, weighted_y[:, i])[0:2]
```

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

```
7499.709075454322
```

Let us compare the RMSE of a MARS model with *mileage, mpg, engineSize* and *year* with a linear regression model having the same predictors.

```
X = train[['mileage','mpg','engineSize','year']]
```

```
model = Earth(max_terms=500, max_degree=2) # note, terms in brackets are the hyperparamete
model.fit(X,y)
print(model.summary())
```

```
C:\Users\akl0407\Anaconda3\lib\site-packages\pyearth\earth.py:813: FutureWarning: `rcond` pa
To use the future default and silence this warning we advise to pass `rcond=None`, to keep us
  pruning_passer.run()

Earth Model
------------------------------------------------------------
Basis Function                      Pruned  Coefficient
------------------------------------------------------------
(Intercept)                         No      -8.13682e+06
h(engineSize-5.5)                   No      9.53908e+06
h(5.5-engineSize)                   Yes     None
h(mileage-21050)                    No      23.4448
h(21050-mileage)                    No      -215.861
h(mileage-21050)*h(5.5-engineSize)  Yes     None
h(21050-mileage)*h(5.5-engineSize)  No      -0.278562
year                                No      4125.85
h(mpg-53.3495)                      Yes     None
h(53.3495-mpg)                      Yes     None
h(mileage-21050)*year               No      -0.0116601
h(21050-mileage)*year               No      0.107624
h(mpg-53.2957)*h(5.5-engineSize)    No      -59801.3
h(53.2957-mpg)*h(5.5-engineSize)    No      59950.5
h(engineSize-5.5)*year              No      -4713.74
h(5.5-engineSize)*year              No      -755.742
h(mileage-1766)*h(53.3495-mpg)      No      -0.00337072
h(1766-mileage)*h(53.3495-mpg)      No      -0.144905
```

```
h(mpg-19.1277)*h(53.3495-mpg)       No      161.153
h(19.1277-mpg)*h(53.3495-mpg)       Yes     None
h(mpg-23.4808)*h(5.5-engineSize)    Yes     None
h(23.4808-mpg)*h(5.5-engineSize)    Yes     None
h(mpg-21.4971)*h(5.5-engineSize)    Yes     None
h(21.4971-mpg)*h(5.5-engineSize)    Yes     None
h(mpg-40.224)*h(5.5-engineSize)     Yes     None
h(40.224-mpg)*h(5.5-engineSize)     No      298.139
engineSize*h(5.5-engineSize)        No      -2553.17
h(mpg-22.2566)                      Yes     None
h(22.2566-mpg)                      No      29257.3
h(mpg-20.7712)*h(22.2566-mpg)       No      143796
h(20.7712-mpg)*h(22.2566-mpg)       No      -1249.17
h(mpg-21.4971)*h(22.2566-mpg)       No      -315486
h(21.4971-mpg)*h(22.2566-mpg)       Yes     None
h(mpg-27.0995)*h(mpg-22.2566)       No      3855.71
h(27.0995-mpg)*h(mpg-22.2566)       Yes     None
h(mpg-29.3902)*year                 No      6.05449
h(29.3902-mpg)*year                 No      -20.176
h(mpg-28.0681)*h(5.5-engineSize)    No      59901.6
h(28.0681-mpg)*h(5.5-engineSize)    No      -55502.2
h(mpg-23.2962)*h(mpg-22.2566)       No      -56126
h(23.2962-mpg)*h(mpg-22.2566)       No      73153.9
h(mpg-69.0719)*h(mpg-53.3495)       Yes     None
h(69.0719-mpg)*h(mpg-53.3495)       No      -124.847
h(engineSize-5.5)*h(22.2566-mpg)    No      -20955.8
h(5.5-engineSize)*h(22.2566-mpg)    No      -8336.23
h(mpg-23.9595)*h(mpg-22.2566)       No      -62983
h(23.9595-mpg)*h(mpg-22.2566)       Yes     None
h(mpg-23.6406)*h(mpg-22.2566)       No      115253
h(23.6406-mpg)*h(mpg-22.2566)       Yes     None
h(mpg-56.1908)                      Yes     None
h(56.1908-mpg)                      No      -2239.85
h(mpg-29.7993)*h(53.3495-mpg)       No      -139.61
h(29.7993-mpg)*h(53.3495-mpg)       No      788.756
------------------------------------------------------------
MSE: 49704412.0771, GCV: 51526765.3943, RSQ: 0.8153, GRSQ: 0.8086

C:\Users\akl0407\Anaconda3\lib\site-packages\pyearth\earth.py:1066: FutureWarning: `rcond` pa
To use the future default and silence this warning we advise to pass `rcond=None`, to keep us
  coef, resid = np.linalg.lstsq(B, weighted_y[:, i])[0:2]
```

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

```
7614.158359050244
```

```
ols_object = smf.ols(formula = 'price~(year+engineSize+mileage+mpg)**2', data = train)
model = ols_object.fit()
pred = model.predict(test)
np.sqrt(mean_squared_error(pred,test.price))
```

```
8729.912066822455
```

The RMSE for the MARS model is lesser than that of the linear regression model, as expected.

3 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
from sklearn.model_selection import cross_val_score,train_test_split
from sklearn.metrics import mean_squared_error,r2_score
from sklearn.model_selection import KFold
from sklearn.tree import DecisionTreeRegressor
from sklearn.model_selection import GridSearchCV, ParameterGrid

#Libraries for visualizing trees
from sklearn.tree import export_graphviz
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

3.1 Building a regression tree

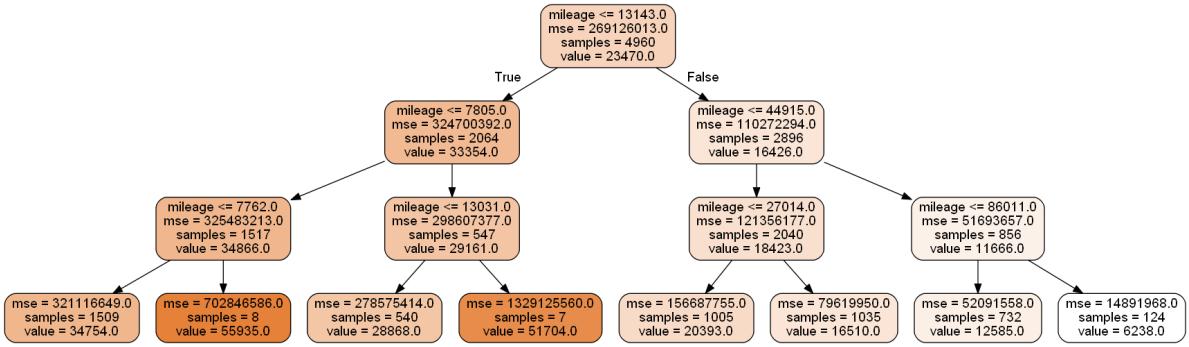
Develop a regression tree to predict car price based on mileage

```
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())
```



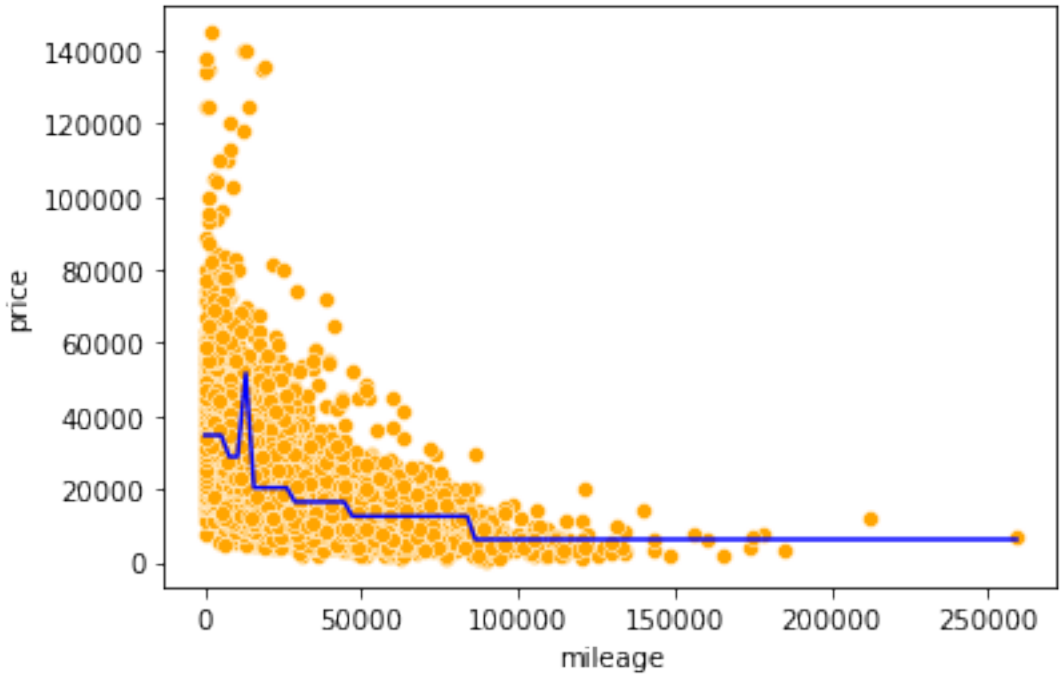

```
#prediction on test data
pred=model.predict(test[['mileage']])
```

```
#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')
```

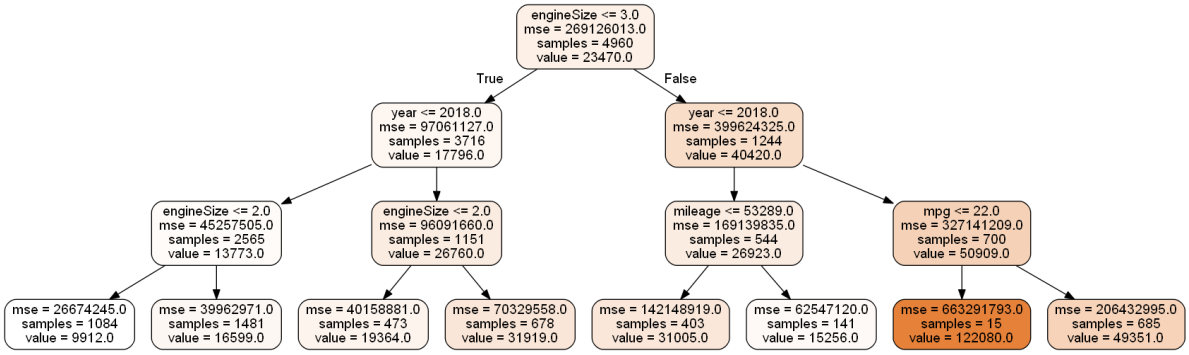
```
<AxesSubplot:xlabel='mileage', ylabel='price'>
```



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())
```



3.2 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.

3.2.1 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
```

3.2.2 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=
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 the mean squared error, 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=
                     scoring=['neg_mean_squared_error', 'r2'], refit = 'neg_mean_squared_er
model.fit(X, y)
print (model.best_score_, model.best_params_)
```

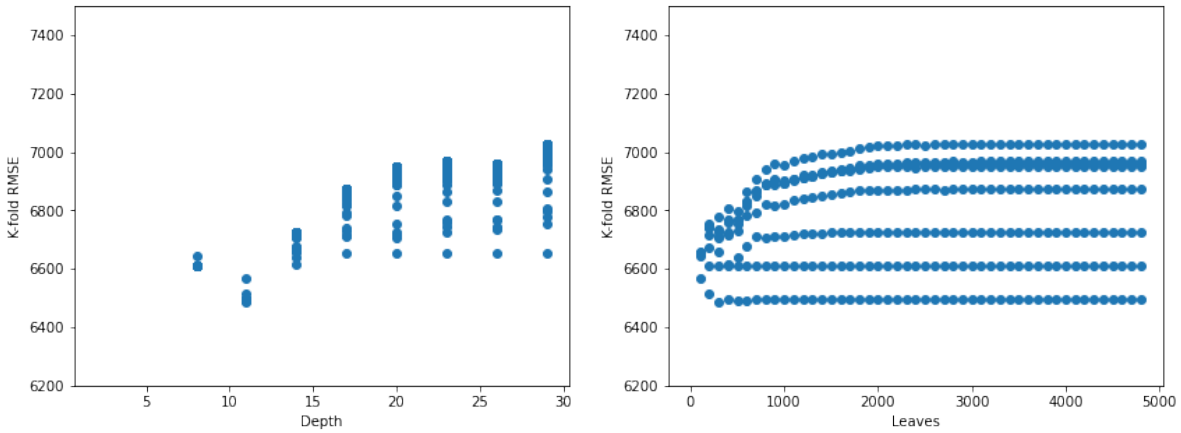
```
Fitting 5 folds for each of 490 candidates, totalling 2450 fits
-42064467.15261547 {'max_depth': 11, 'max_leaf_nodes': 302}
```

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

```
fig, axes = plt.subplots(1,2,figsize=(14,5))
plt.subplots_adjust(wspace=0.2)
axes[0].plot(cv_results.param_max_depth, np.sqrt(-cv_results.mean_test_neg_mean_squared_er
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, np.sqrt(-cv_results.mean_test_neg_mean_squar
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.

3.2.3 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=
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
0.8465176078797111 {'max_depth': 10, 'max_leaf_nodes': 262}
Time taken = 1 minutes
```

From the above cross-validation, the optimal hyperparameter values are `max_depth = 10` and `max_leaf_nodes = 262`.

```
#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 and spline regression models (including MARS), with these four predictors. This may be probably due to car price having a highly non-linear association with the predictors.

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.DecisionTreeRegres*

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.

3.3 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 α. All the non-terminal nodes for which α_{eff} is smaller that the optimal α 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

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

```
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':alpha
                    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.

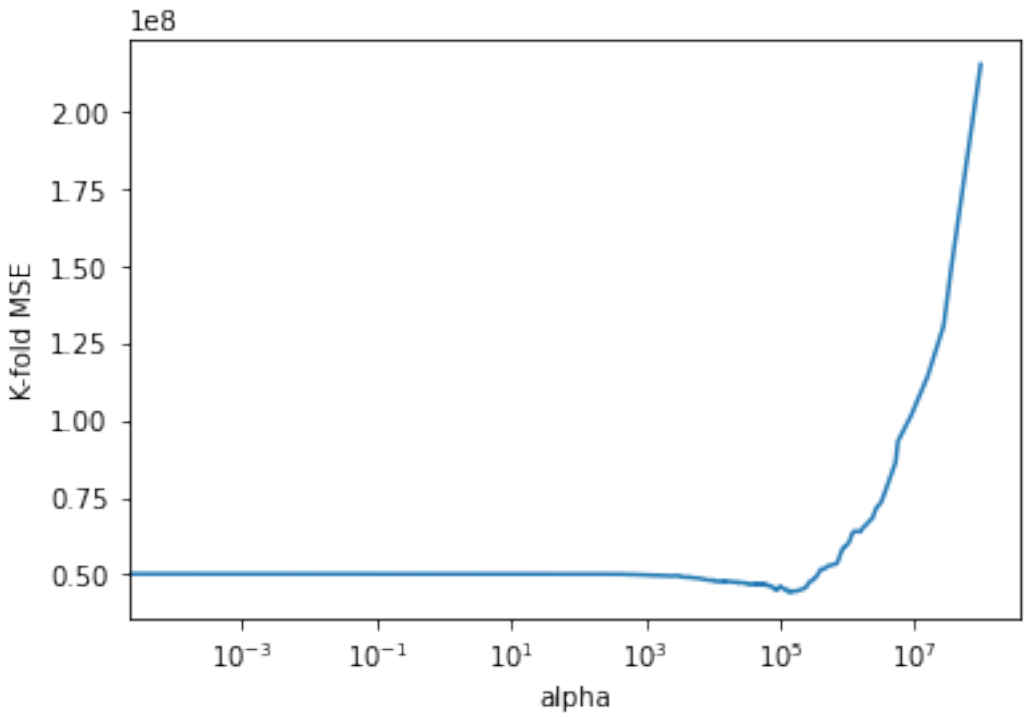
```
model = DecisionTreeRegressor(ccp_alpha=143722.94076639024,random_state=1)
model.fit(X, y)
pred = model.predict(Xtest)
np.sqrt(mean_squared_error(test.price, pred))
```

```
7306.592294294368
```

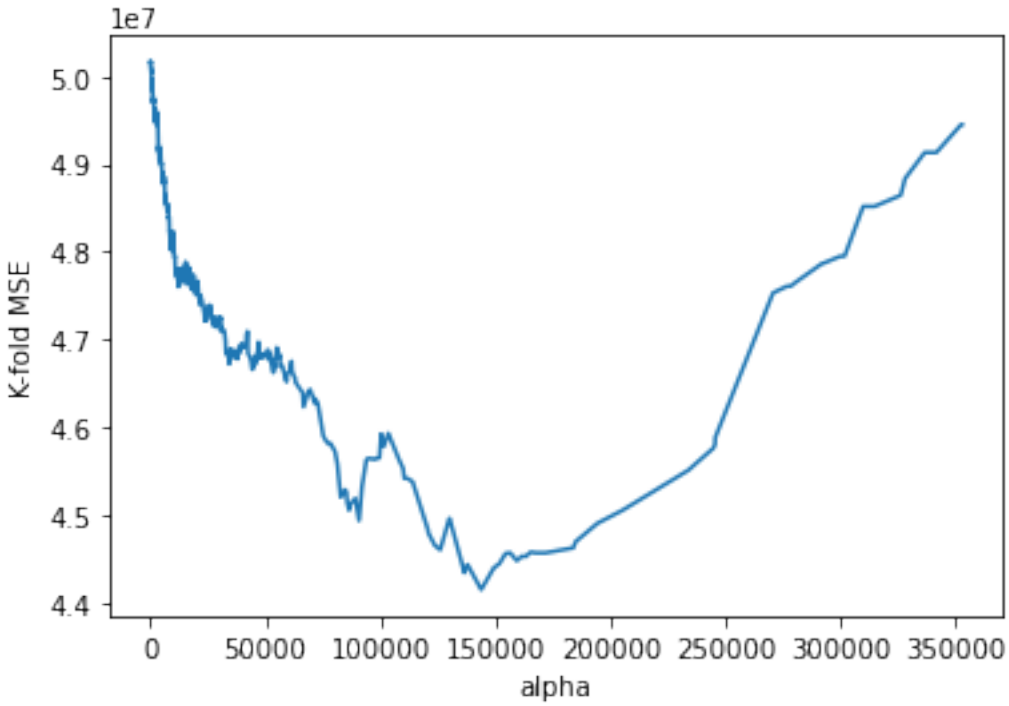
The RMSE for the decision tree with cost complexity pruning is lower than that of linear regression models and spline regression models (including MARS), with these four predictors. However, it is higher than the one obtained with tuning tree parameters using grid search (shown previously). Cost complexity 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 err
plt.plot(alphas[0:4093],cv_error[0:4093])
plt.xlabel('alpha')
plt.ylabel('K-fold MSE');
```



3.3.1 Depth vs alpha; Node counts vs alpha

```
stime = 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)
```

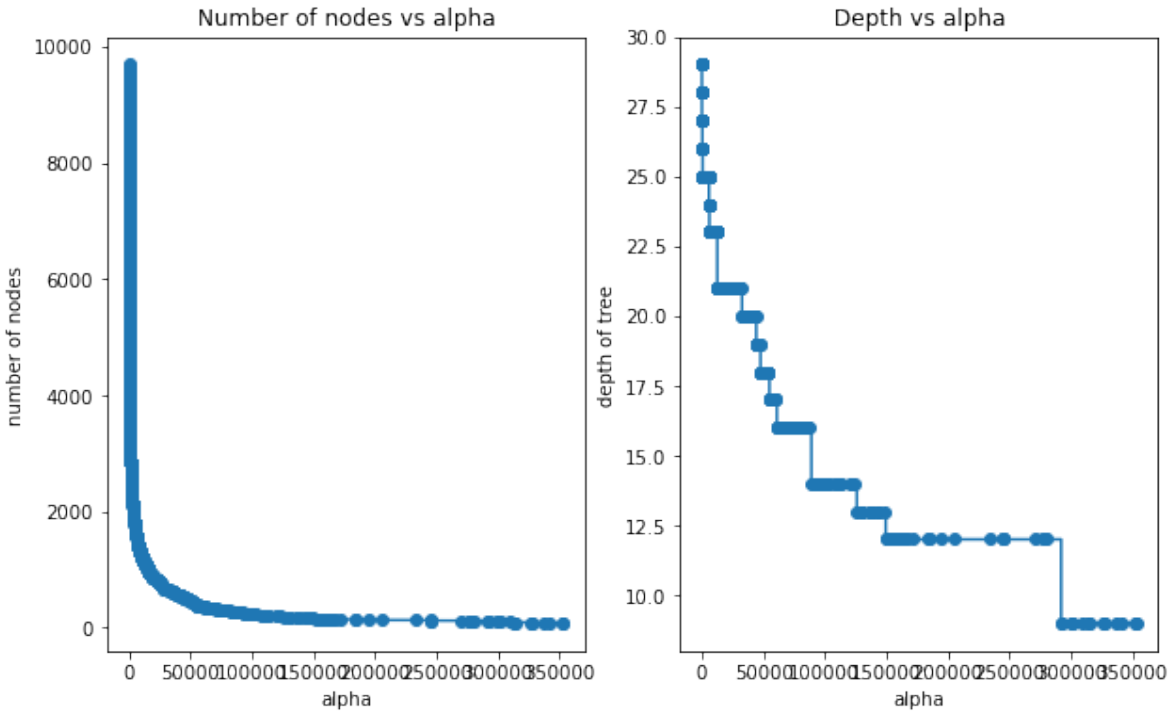
```
268.10325384140015
```

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")#Plotti
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
ax[1].set_xlabel("alpha")
ax[1].set_ylabel("depth of tree")
ax[1].set_title("Depth vs alpha")
#fig.tight_layout()
```

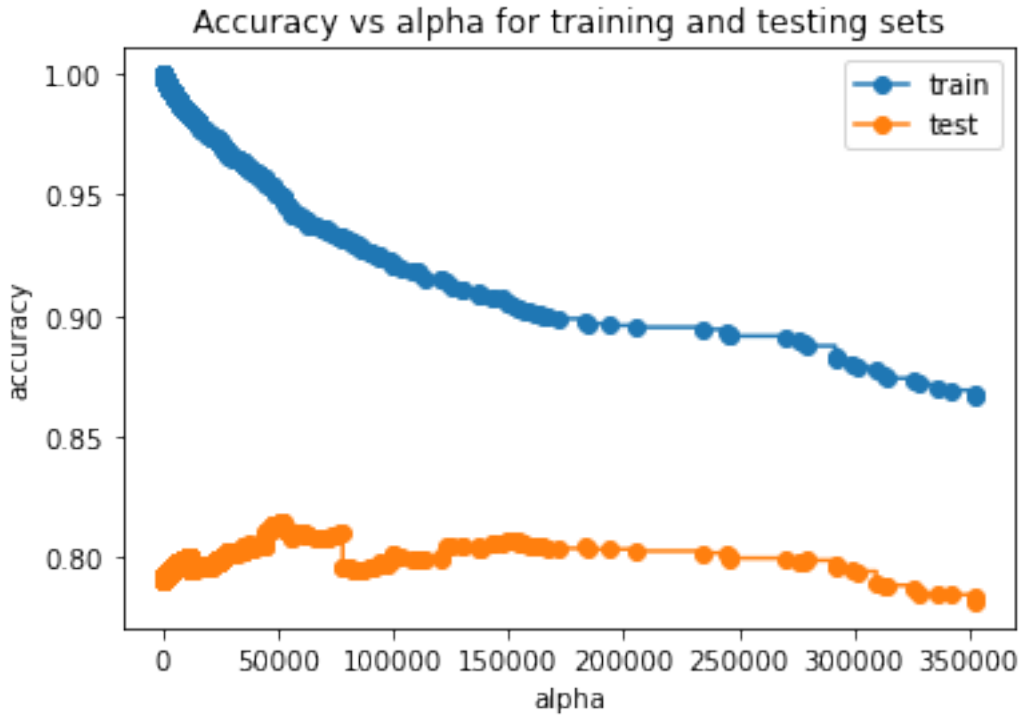
```
Text(0.5, 1.0, 'Depth vs alpha')
```



3.3.2 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-
ax.plot(alphas[0:4093], test_scores[0:4093], marker="o", label="test", drawstyle="steps-po
ax.legend()
plt.show()
```



4 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_sc
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

4.1 Building a classification tree

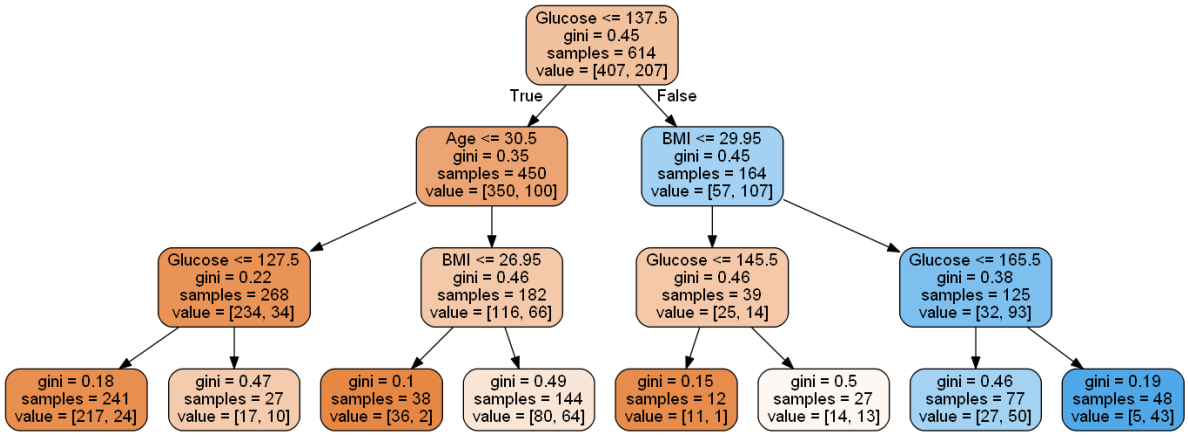
Develop a classification tree to predict if a person has diabetes.

```
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)
```

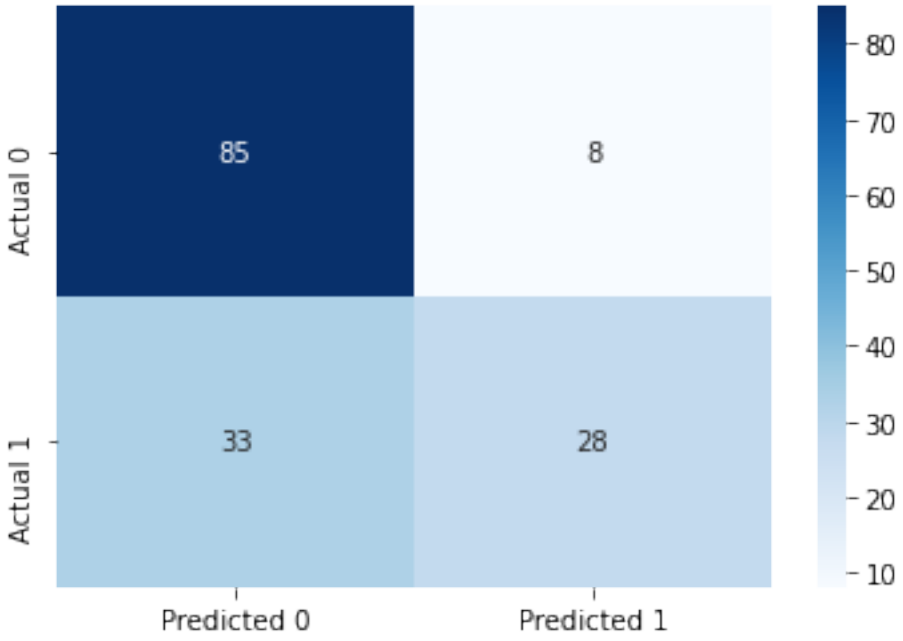
```
DecisionTreeClassifier(max_depth=3, random_state=1)
```

```
#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
```



4.2 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 nods (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 sampl

#Minimizing FNR is equivalent to maximizing recall
grid_search = GridSearchCV(DecisionTreeClassifier(random_state=1), param_grid, scoring=['p
                           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
```

4.3 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.

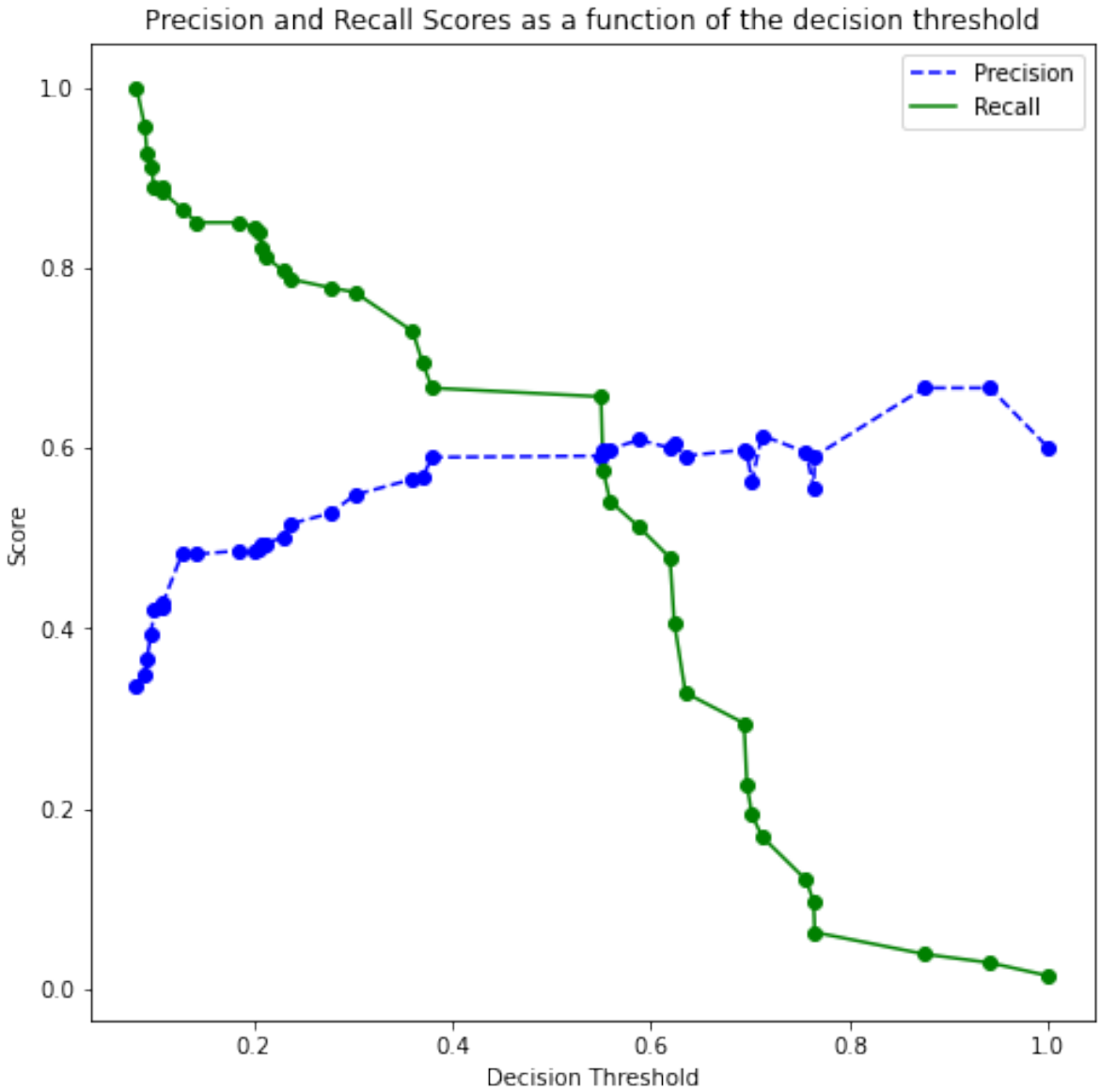
As the recall will always be 100% for a decision threshold probability of zero, we'll find a decision threshold probability that balances recall with another performance metric such as precision, false positive rate, accuracy, etc. Below are a couple of examples that show we can balance recall with (1)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

4.3.1 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_featur

# Note that we are using the cross-validated predicted probabilities, instead of directly
# predicted probabilities on train data, as the model may be overfitting on the train data
# 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),
                                                    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)], axi
```

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

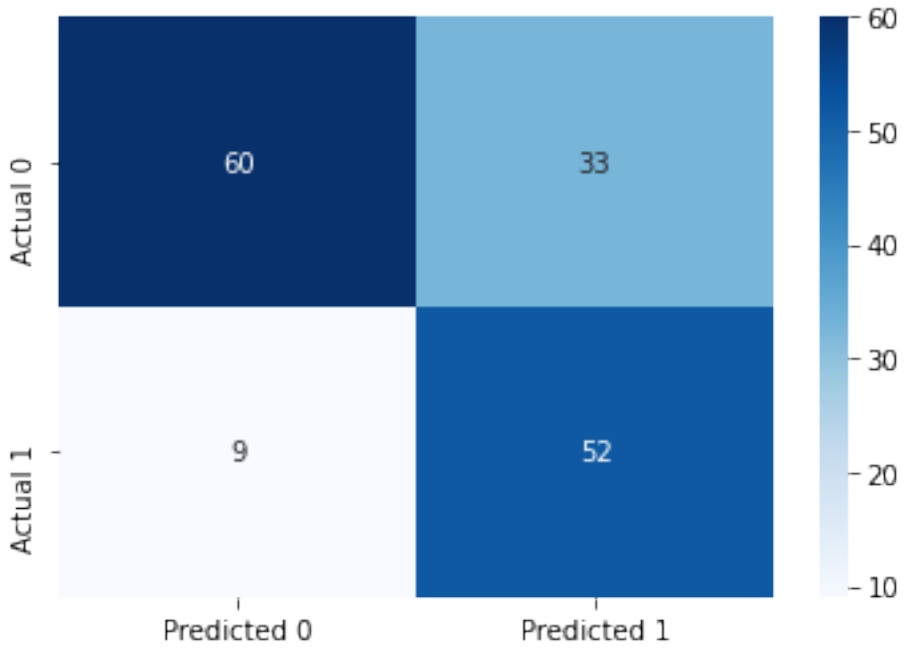
```
[0.1       , 0.42105263, 0.88888889],
[0.10810811, 0.42298851, 0.88888889],
[0.10869565, 0.42857143, 0.88405797],
[0.12820513, 0.48378378, 0.8647343 ],
[0.14285714, 0.48219178, 0.85024155],
[0.18518519, 0.48618785, 0.85024155],
[0.2       , 0.48611111, 0.84541063],
[0.20512821, 0.48876404, 0.84057971],
[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 g
# 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
```



4.3.2 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 sampl

#Minimizing FNR is equivalent to maximizing recall
grid_search = GridSearchCV(DecisionTreeClassifier(random_state=1), param_grid, scoring=['p
                           'roc_auc'], refit="roc_auc", cv=skf, n_jobs=-1, verbose = True
```

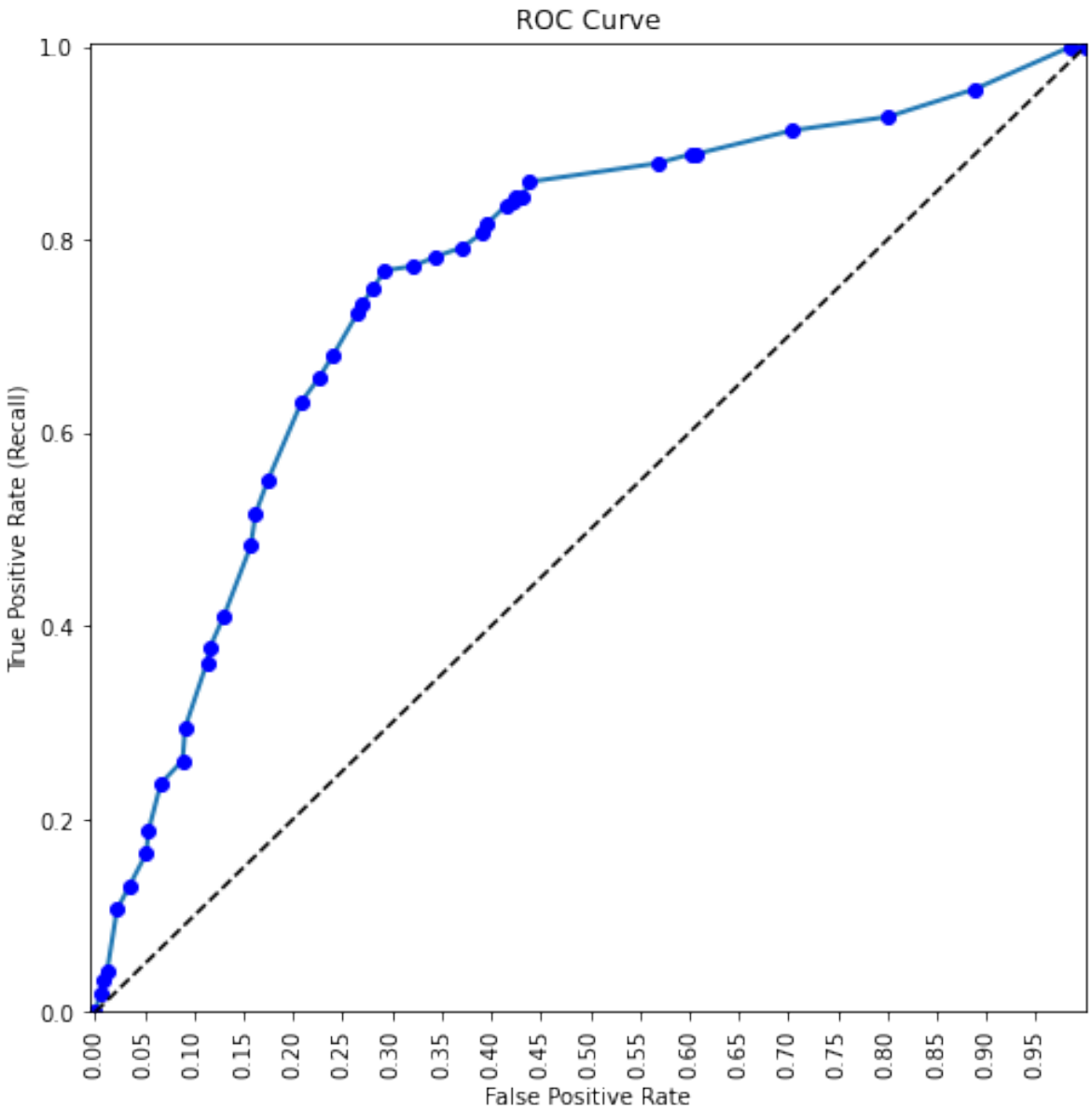
```
grid_search.fit(X, y)

# 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_featur
```

```
cross_val_ypred = cross_val_predict(DecisionTreeClassifier(random_state=1, max_depth = 6,
                                                           max_leaf_nodes=9, max_features=
                                                    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.resh
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 de
# the first value will provide the maximum threshold probability for the recall to be more
# We wish to find the maximum threshold probability to obtain the minimum possible FPR
recall_more_than_80[0]
```

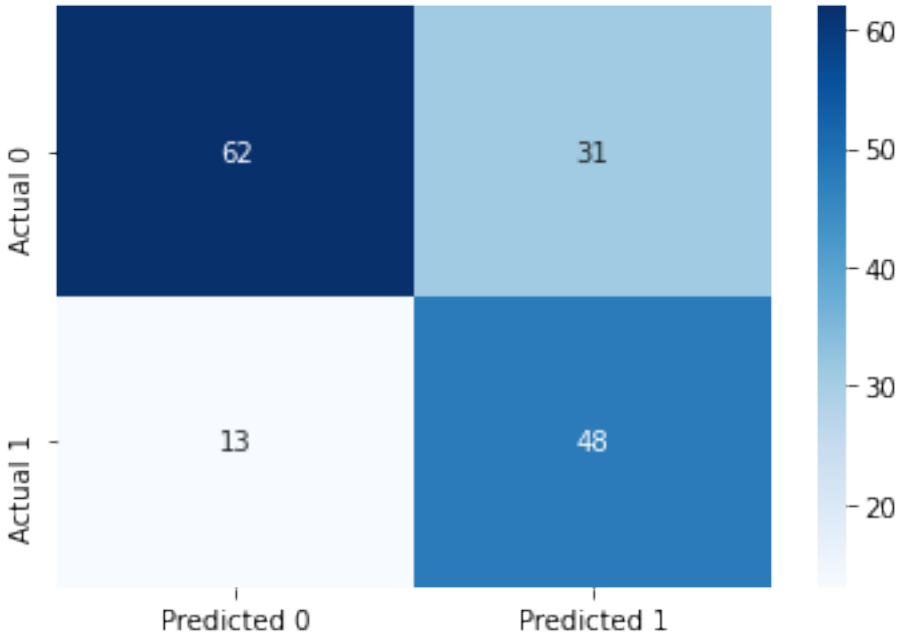
```
array([0.21276596, 0.80676329, 0.39066339])
```

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 g
# 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
```



4.4 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 Cos
```

```
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_al
                                                   scoring=['precision','recall','accuracy'
                                                   refit="recall", cv=skf, n_jobs=-1, verbo
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.

5 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.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
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_sc
accuracy_score, precision_score, confusion_matrix, mean_squared_error, r2_score

#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
```

```
#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']
```

5.1 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(base_estimator=DecisionTreeRegressor(), n_estimators=10, random_s
                         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.

5.1.1 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
oob_rsquared={};test_rsquared={};oob_rmse={};test_rmse = {}
for i in np.linspace(10,400,40,dtype=int):
    model = BaggingRegressor(base_estimator=DecisionTreeRegressor(), n_estimators=i, rando
                        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))
```

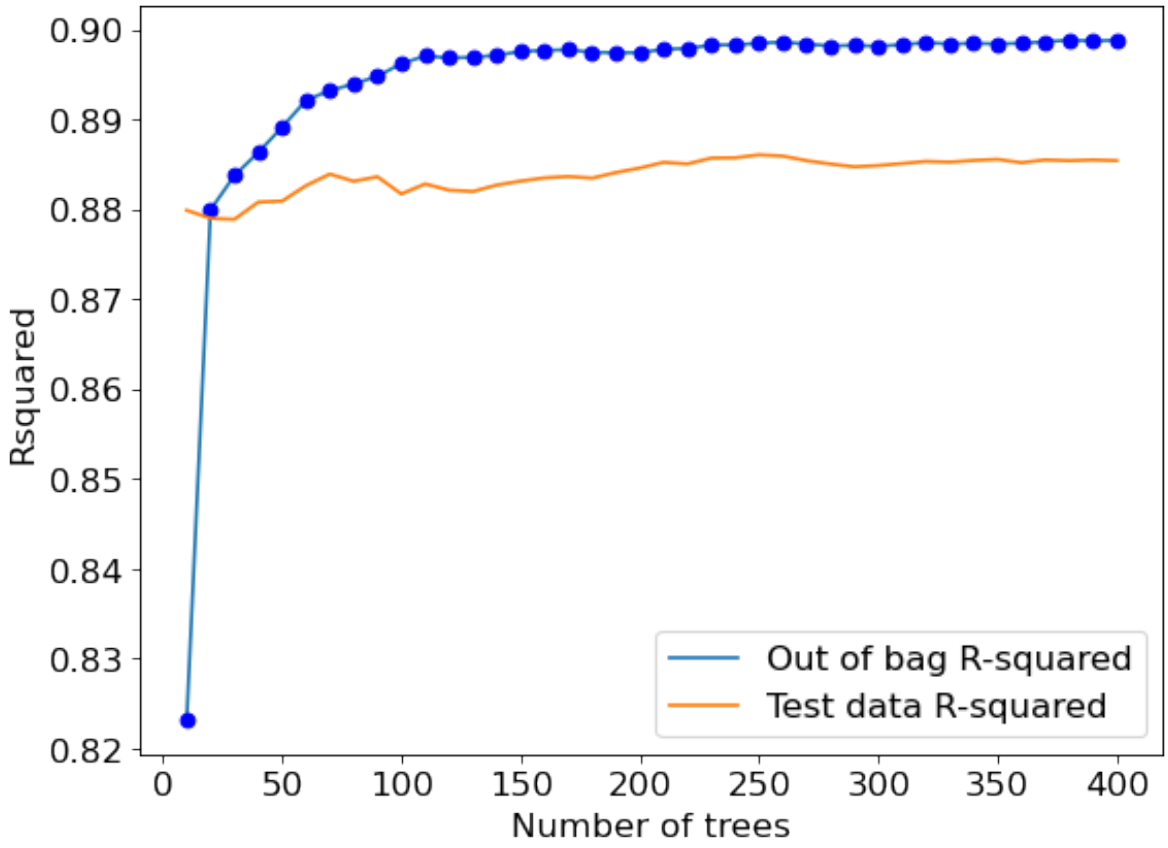
```
C:\Users\akl0407\Anaconda3\lib\site-packages\sklearn\ensemble\_bagging.py:1069: UserWarning:
  warn("Some inputs do not have OOB scores. "
C:\Users\akl0407\Anaconda3\lib\site-packages\sklearn\ensemble\_bagging.py:1069: UserWarning:
  warn("Some inputs do not have OOB scores. "
```

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()
```

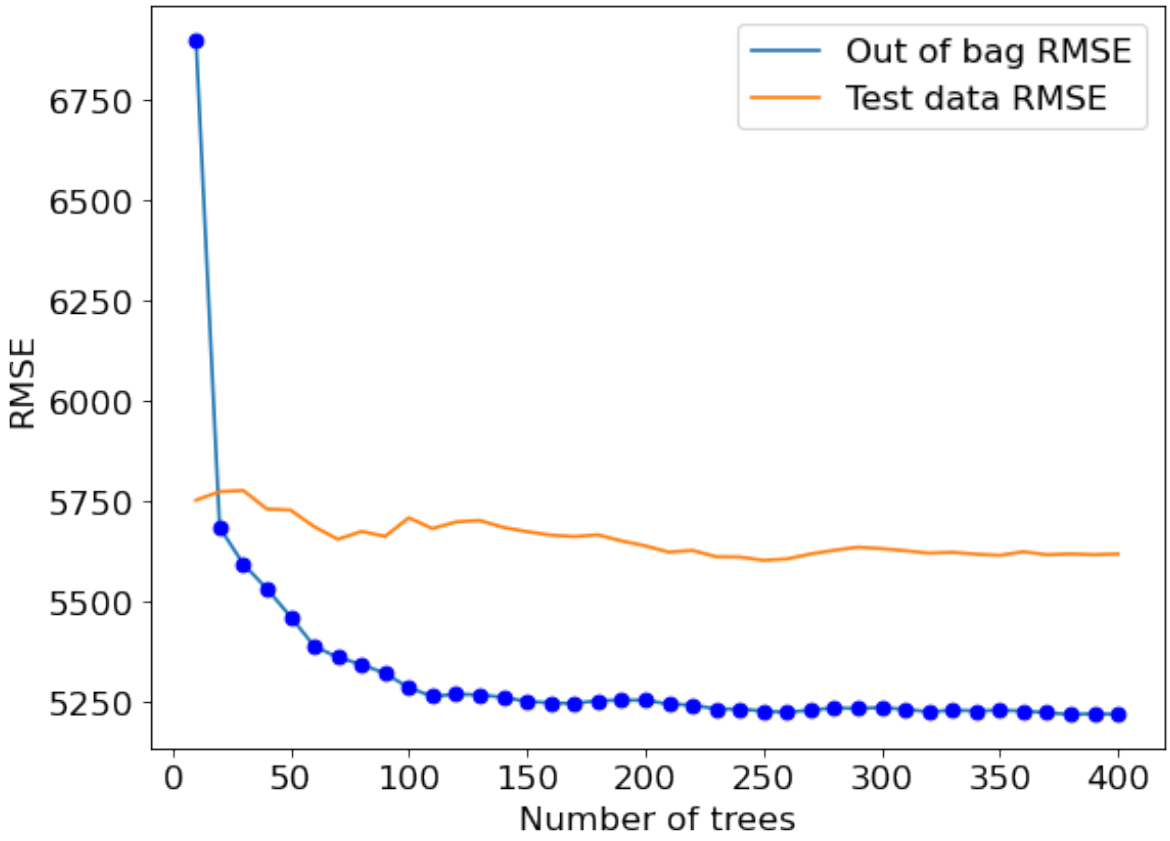
```
<matplotlib.legend.Legend at 0x26f886f1610>
```



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()
```

```
<matplotlib.legend.Legend at 0x13712485df0>
```



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_
                         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
```

5.1.2 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 neglible.

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

params = {'base_estimator': [DecisionTreeRegressor(random_state = 1),LinearRegression()],#
          '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': Tru
```

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_
                         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
```

5.2 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')
```

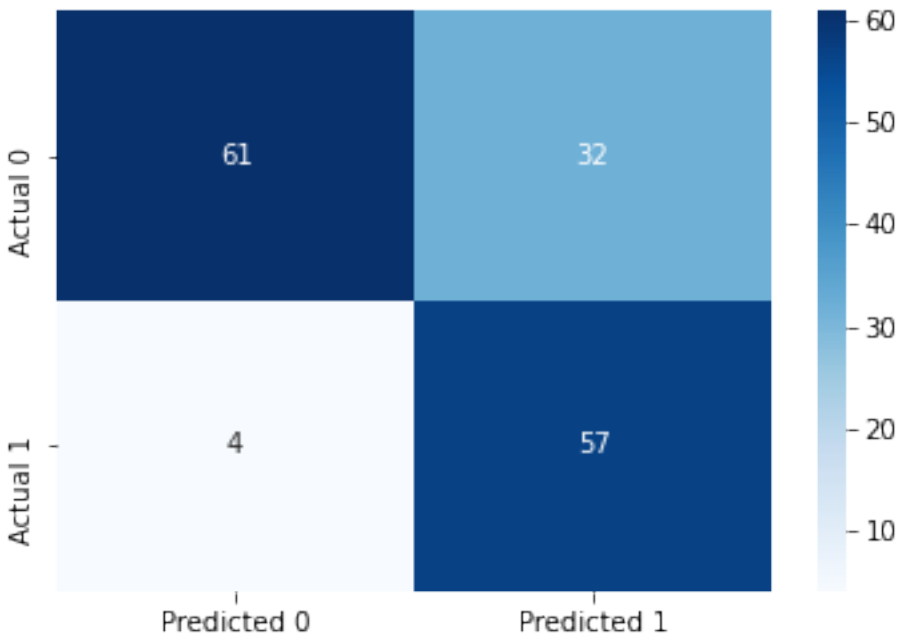
```
X = train.drop(columns = 'Outcome')
Xtest = test.drop(columns = 'Outcome')
y = train['Outcome']
ytest = test['Outcome']
```

```
#Bagging the results of 10 decision trees to predict car price
model = BaggingClassifier(base_estimator=DecisionTreeClassifier(), n_estimators=150, rando
                         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 g
# 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 probabiltiy 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.

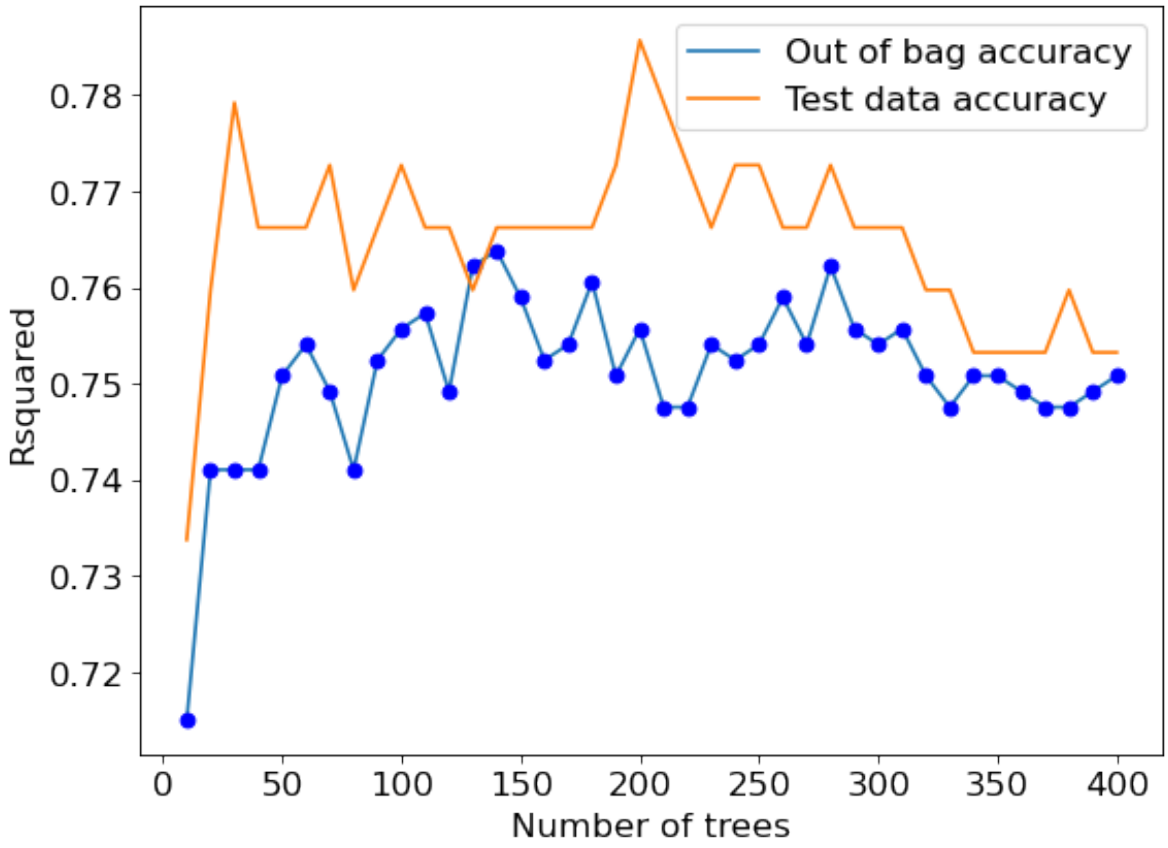
5.2.1 Model accuracy vs number of trees

```
#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, ran
                      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: S
  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()
```

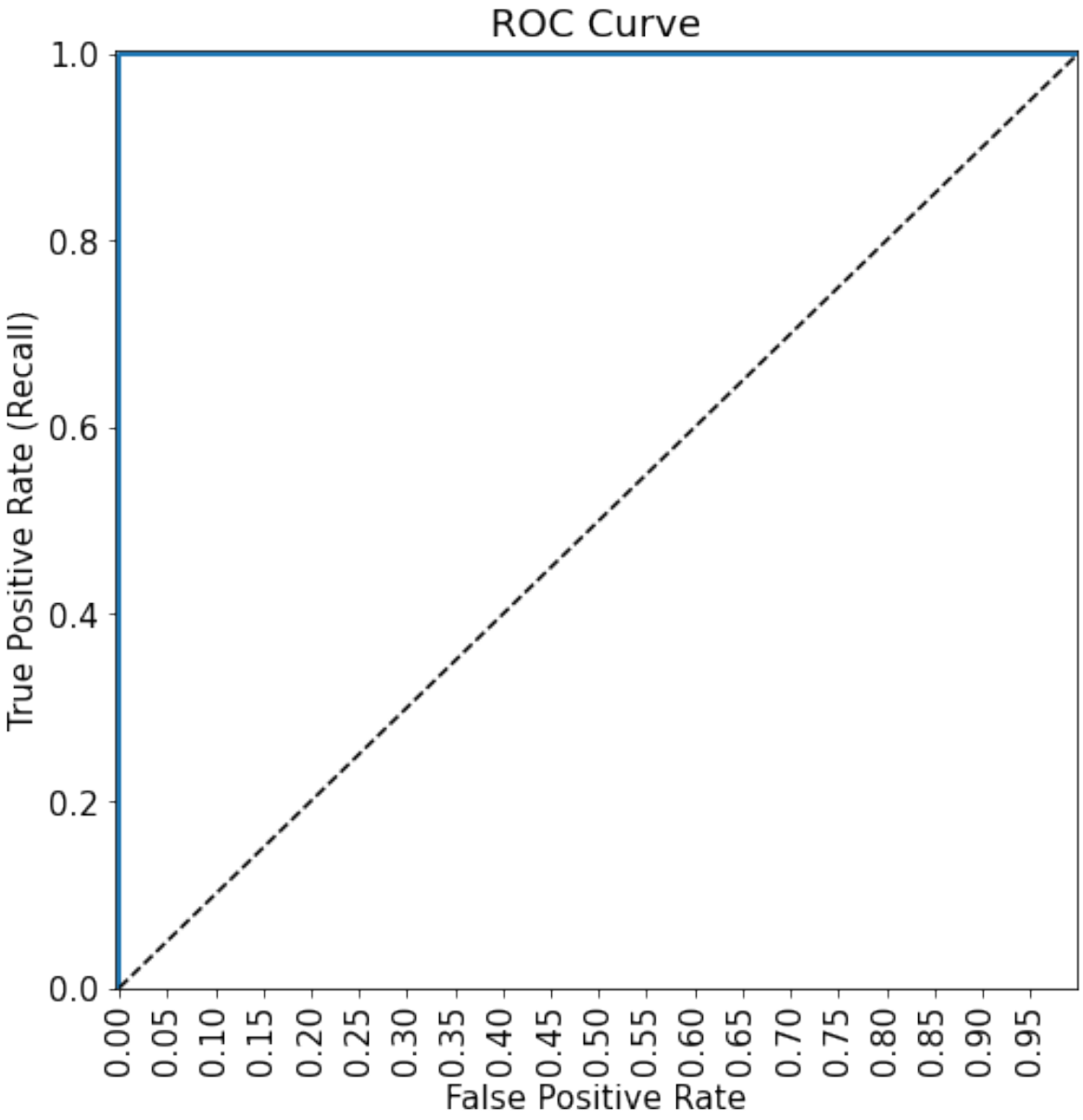
```
<matplotlib.legend.Legend at 0x223302e62e0>
```




```
#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)
```

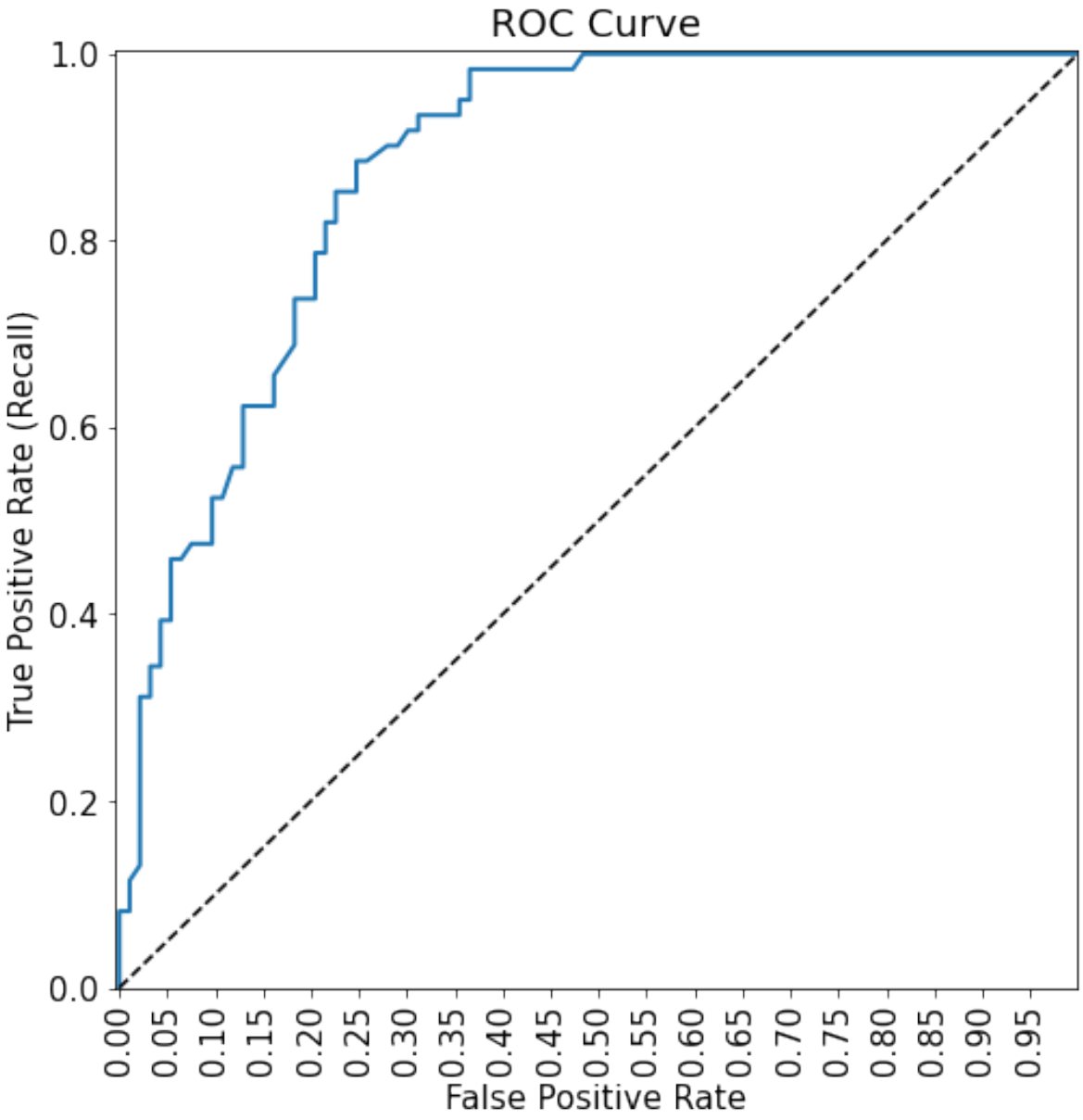
```
1.0
```



Note that there is perfect separation in train data as 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
```



5.2.2 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': T
```

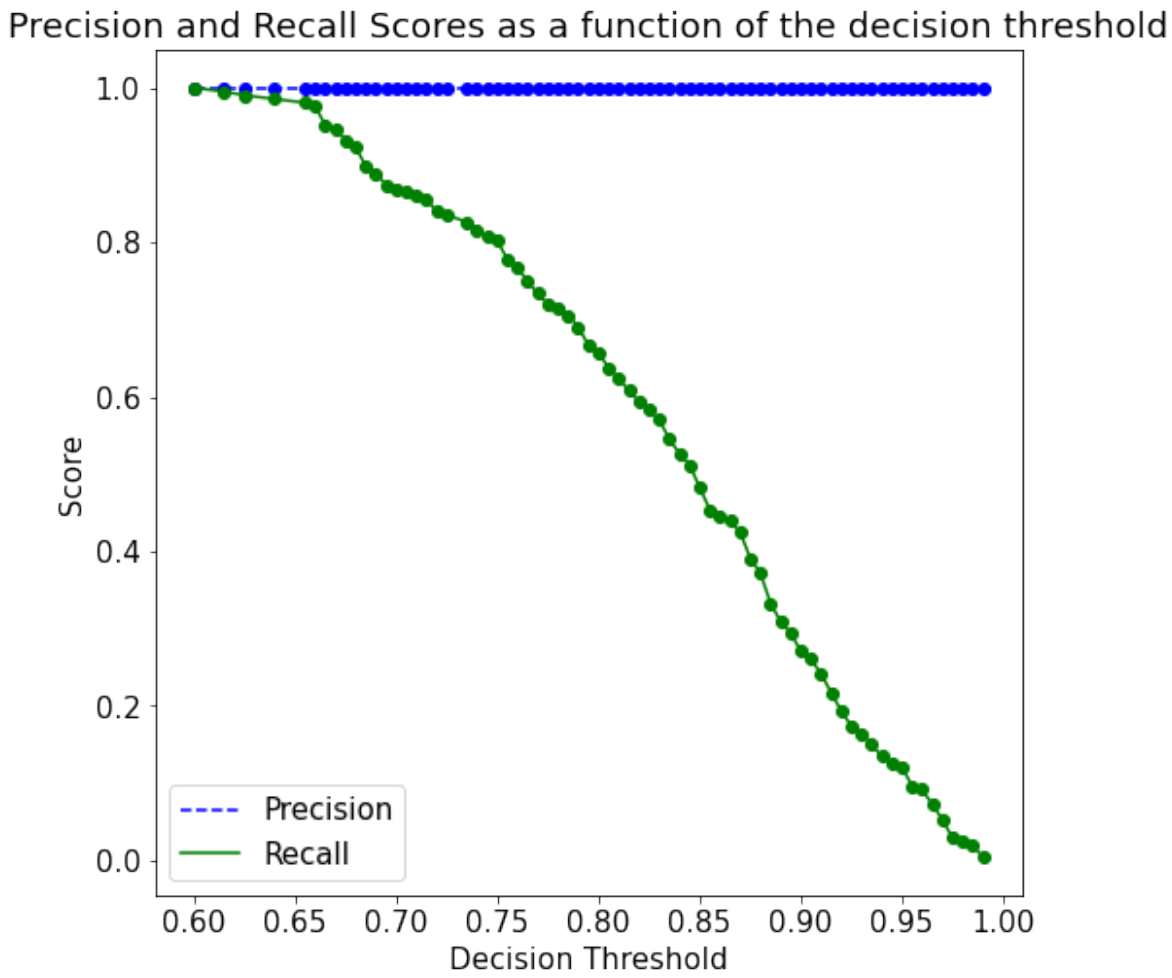
5.2.3 Tuning the decision threshold probability

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

```
model = BaggingClassifier(base_estimator=DecisionTreeClassifier(random_state=1), n_estimat
                          random_state=1,max_features=1.0, oob_score=True,
                         max_samples=1.0,n_jobs=-1,bootstrap=True,bootstrap_features=False)
```

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.

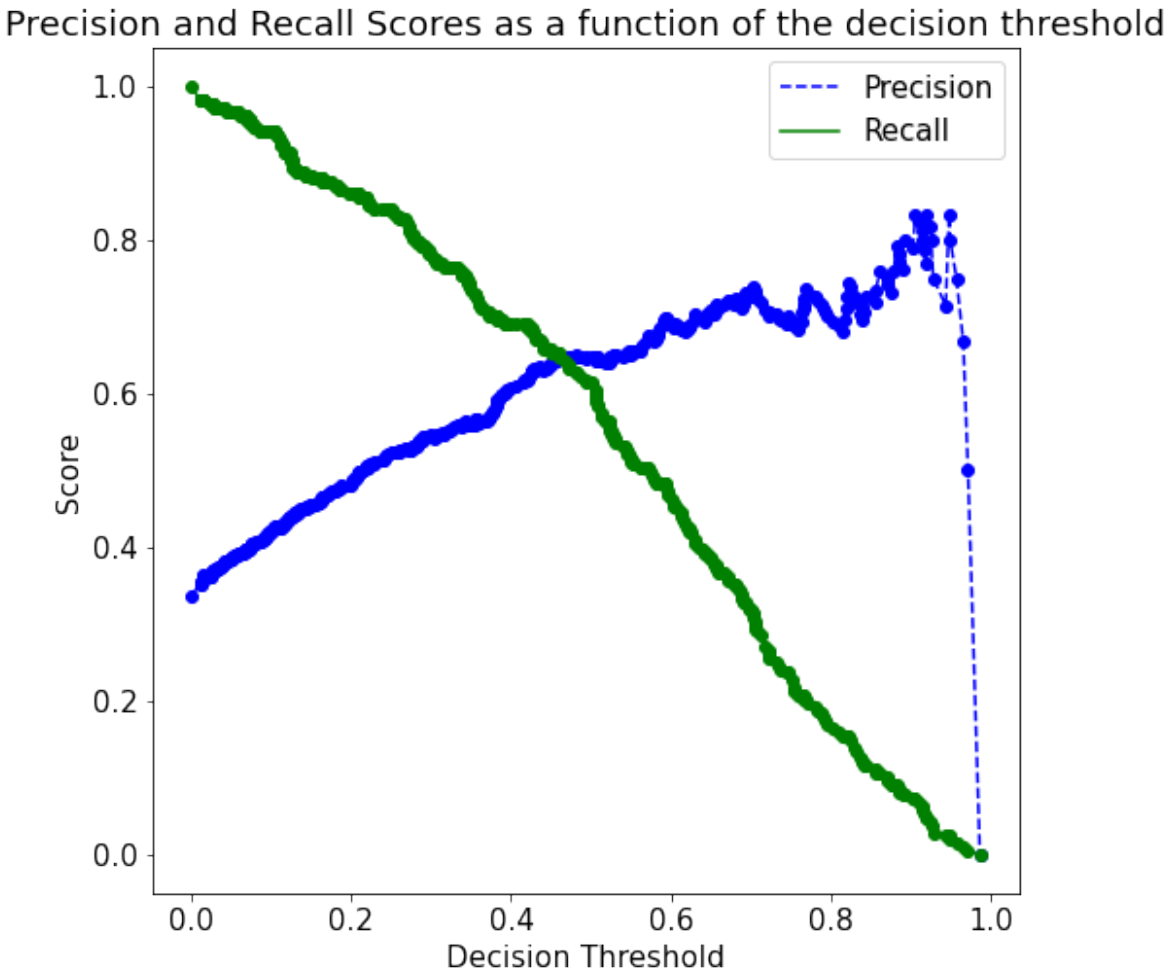
```
ypred = model.predict_proba(X)[:,1]
p, r, thresholds = precision_recall_curve(y, ypred)
def plot_precision_recall_vs_threshold(precisions, recalls, thresholds):
    plt.figure(figsize=(8, 8))
    plt.title("Precision and Recall Scores as a function of the decision threshold")
    plt.plot(thresholds,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



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)
```



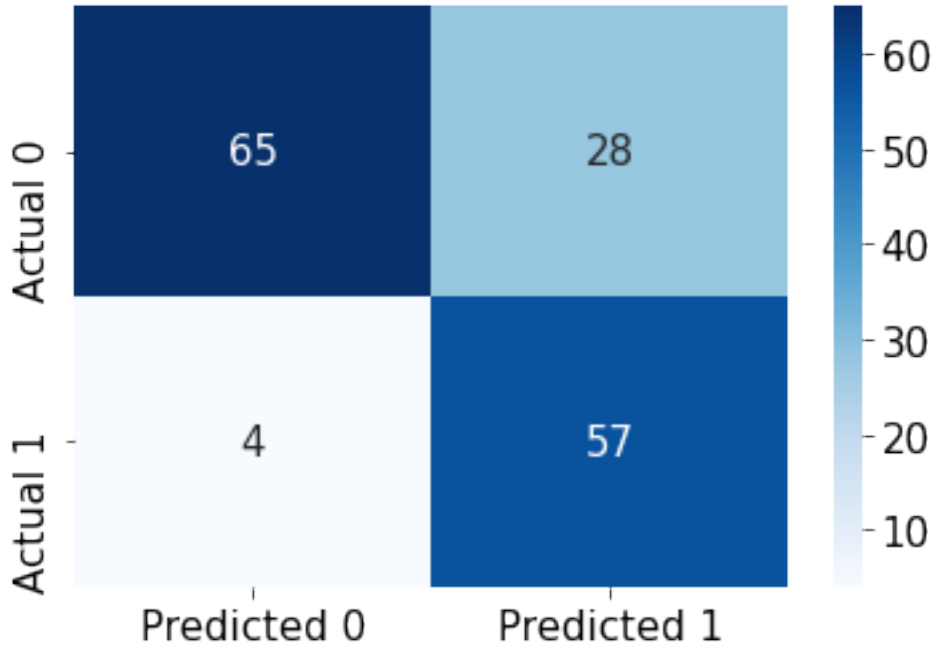

```
# Thresholds with precision and recall
all_thresholds = np.concatenate([thresholds.reshape(-1,1), p[:-1].reshape(-1,1), r[:-1].re
recall_more_than_80 = all_thresholds[all_thresholds[:,2]>0.8,:]
# As the values in 'recall_more_than_80' are arranged in decreasing order of recall and in
# the last value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

6 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,Rand
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_sc
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
```

```
#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()
```

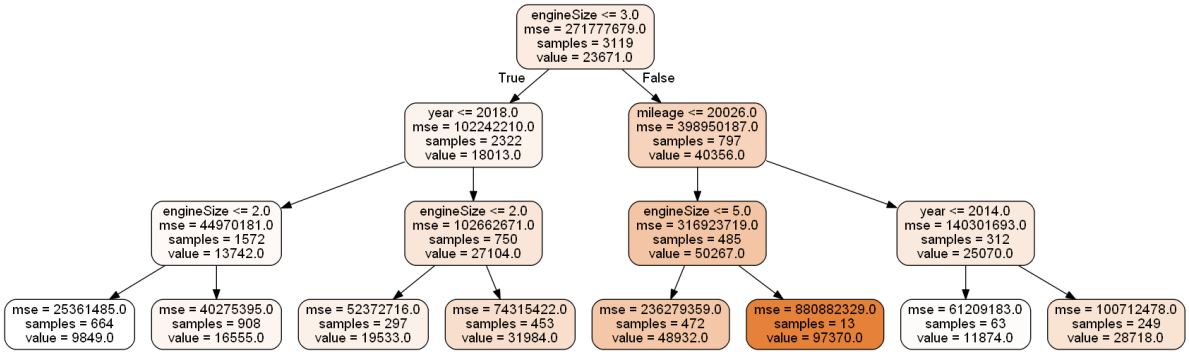
	carID	brand	model	year	transmission	mileage	fuelType	tax	mpg	engineSize	price
0	18473	bmw	6 Series	2020	Semi-Auto	11	Diesel	145	53.3282	3.0	37980
1	15064	bmw	6 Series	2019	Semi-Auto	10813	Diesel	145	53.0430	3.0	33980
2	18268	bmw	6 Series	2020	Semi-Auto	6	Diesel	145	53.4379	3.0	36850
3	18480	bmw	6 Series	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(base_estimator=DecisionTreeRegressor(max_depth=3), n_estimators=1
                         n_jobs=-1).fit(X, y)
```

```
#Change the index of model.estimators_[index] to visualize the 10 bagged trees, one at a t
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.

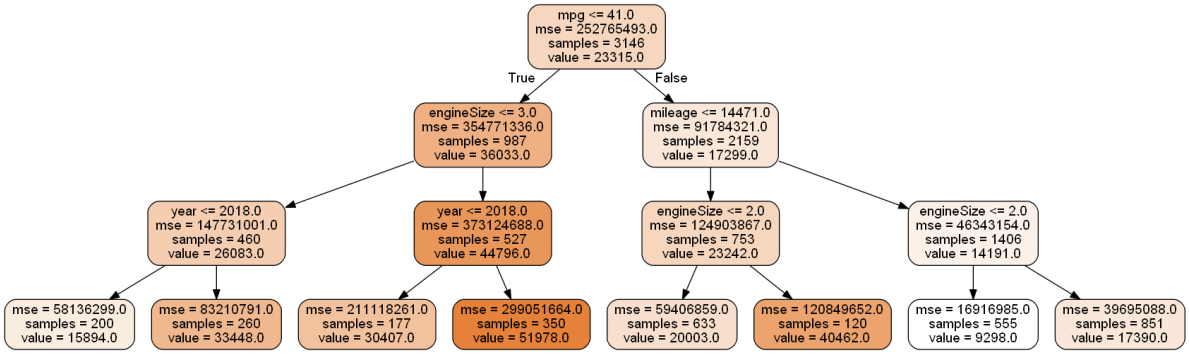
6.1 Random Forest for regression

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 predicto
#to split, to predict car price
model = RandomForestRegressor(n_estimators=10, random_state=1,max_features="sqrt",max_dept
                        n_jobs=-1).fit(X, y)
```

```
#Change the index of model.estimators_[index] to visualize the 10 random forest trees, one
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 predicto
#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.

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
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))
```

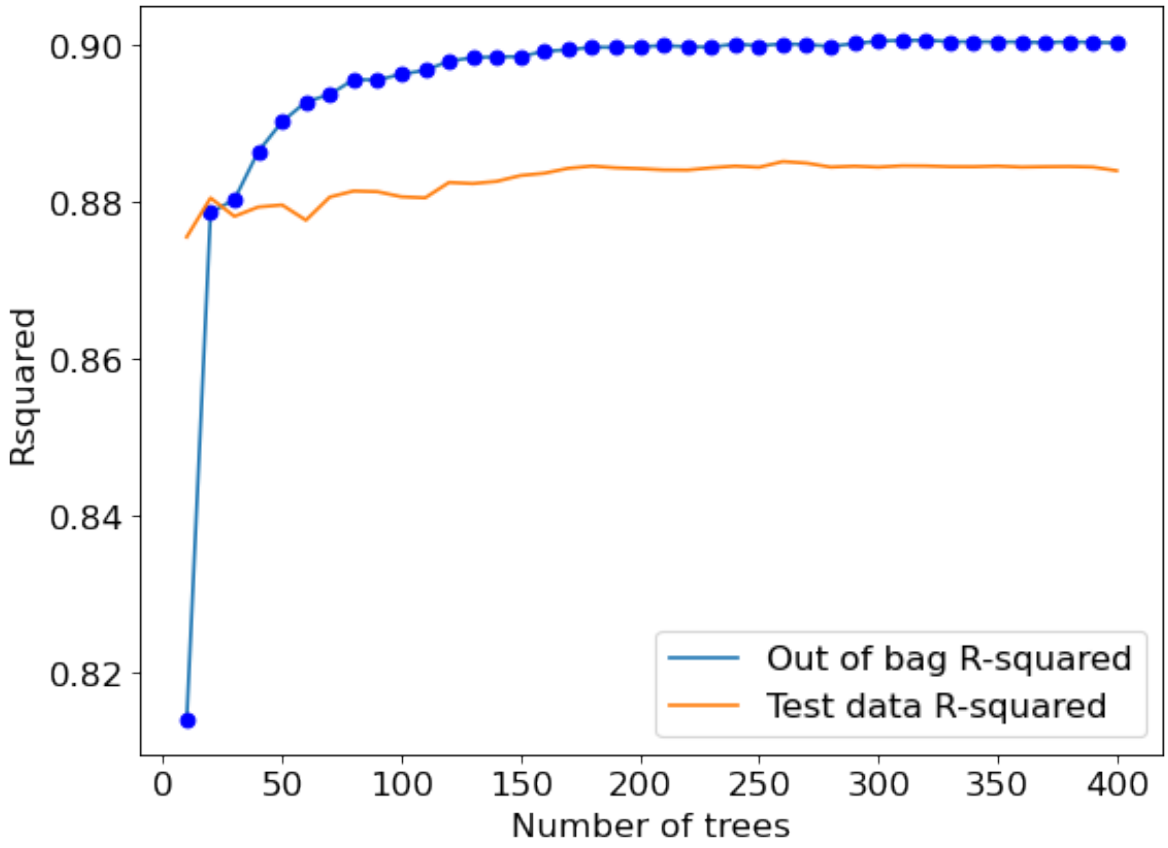
```
C:\Users\akl0407\Anaconda3\lib\site-packages\sklearn\ensemble\_forest.py:833: UserWarning: So
  warn("Some inputs do not have OOB scores. "
C:\Users\akl0407\Anaconda3\lib\site-packages\sklearn\ensemble\_forest.py:833: UserWarning: So
  warn("Some inputs do not have OOB scores. "
```

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()
```

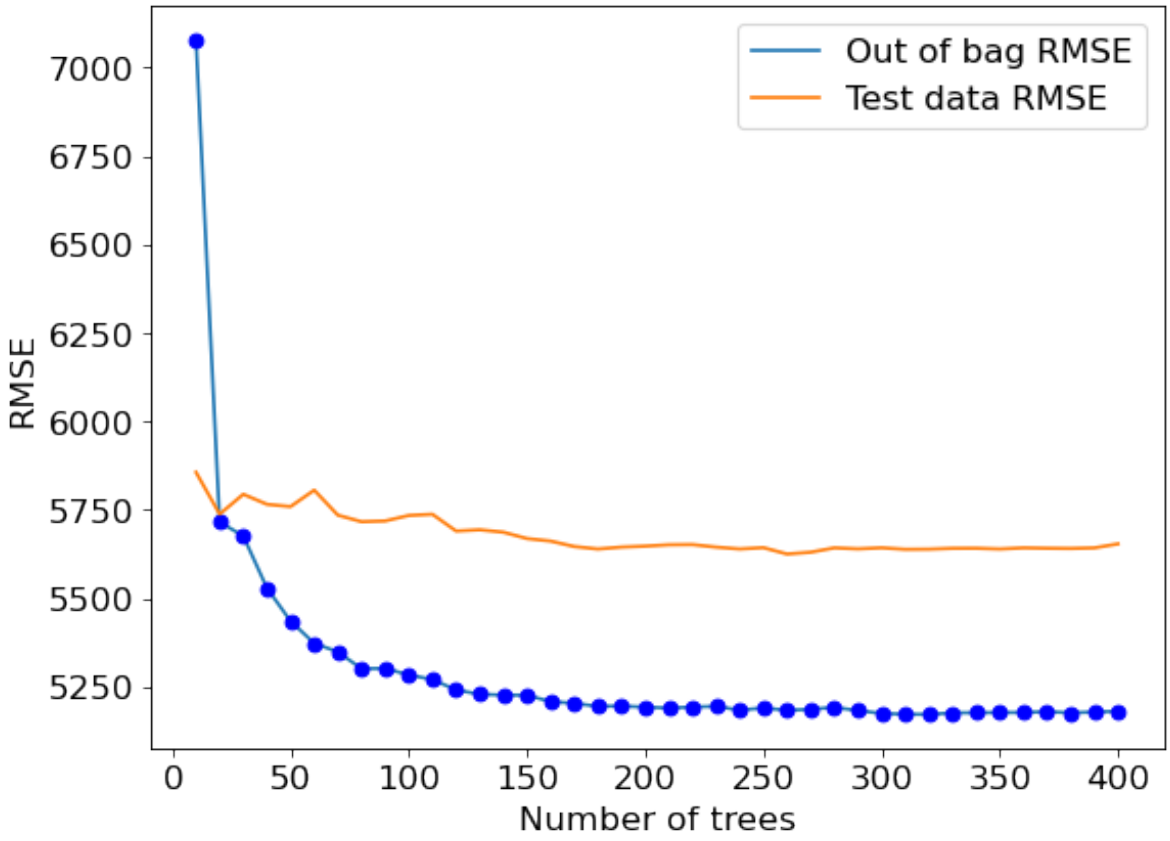
```
<matplotlib.legend.Legend at 0x17677ce6580>
```



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()
```

```
<matplotlib.legend.Legend at 0x1767fff7460>
```



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
```

6.1.2 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.

```
#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
#5-fold cross validation
start_time = time.time()

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

params = {'n_estimators': [300],
          'max_depth': [12,15,18],
          'max_leaf_nodes':[1100,1200,1300],
          '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
                                  max_depth=pr[1],
                                   max_leaf_nodes=pr[2],max_features=pr[3],
                                   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("Best params = ", param_list[np.argmax(oob_score)])
print("Best score (R-squared) = ", np.max(oob_score))
```

```
time taken =  0.3812573711077372  minutes
Best params =  (300, 15, 1200, 3)
Best score (R-squared) =  0.9018153771851661
```

There is a very small increase in OOB R-squared after pruning the individual trees.

```
#Model with optimal parameters
model = RandomForestRegressor(n_estimators=300, random_state=1,max_leaf_nodes=1200,max_dep
                        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)))
```

```
5654.435430858449
```

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, we'll see in the assignment that sometimes individual pruning of trees may be useful.

```
#Tuning only n_estimators and max_features produces similar results
start_time = time.time()
params = {'n_estimators': [300],
          '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
                                  max_features=pr[1],      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("Best params = ", param_list[np.argmax(oob_score)])
print("Best score (R-squared) = ", np.max(oob_score))
```

```
time taken =  0.04980873266855876  minutes
Best params =  (300, 2)
Best score (R-squared) =  0.9005106776136418
```

```
#Model with optimal parameters
model = RandomForestRegressor(n_estimators=300, random_state=1,
                          n_jobs=-1, max_features=2).fit(X, y)
np.sqrt(mean_squared_error(test.price, model.predict(Xtest)))
```

```
5642.45839697972
```

6.2 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']
```

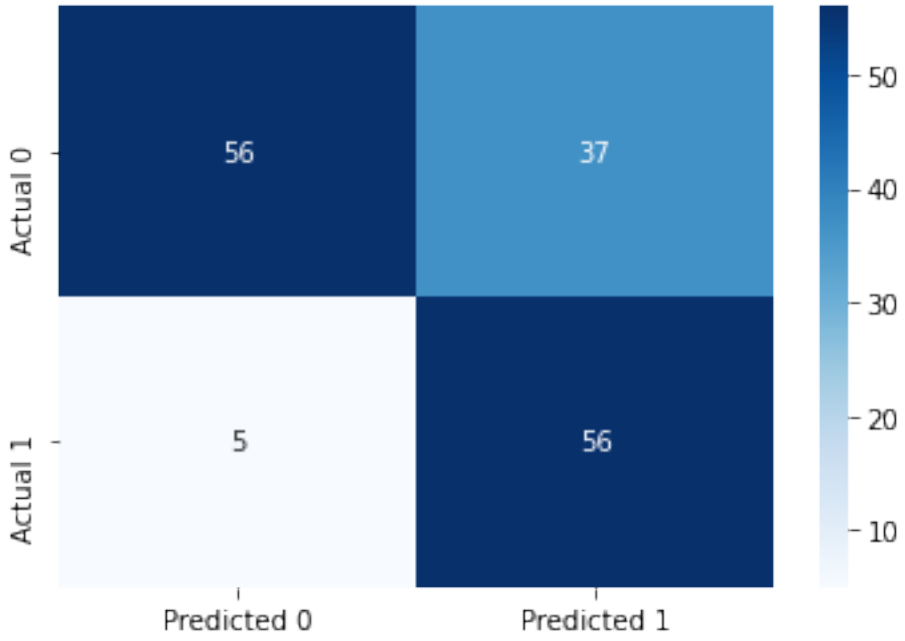
```
#Ensembling the results of 10 decision trees
model = RandomForestClassifier(n_estimators=200, random_state=1,max_features="sqrt",n_jobs
```

```
#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 g
# 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
```



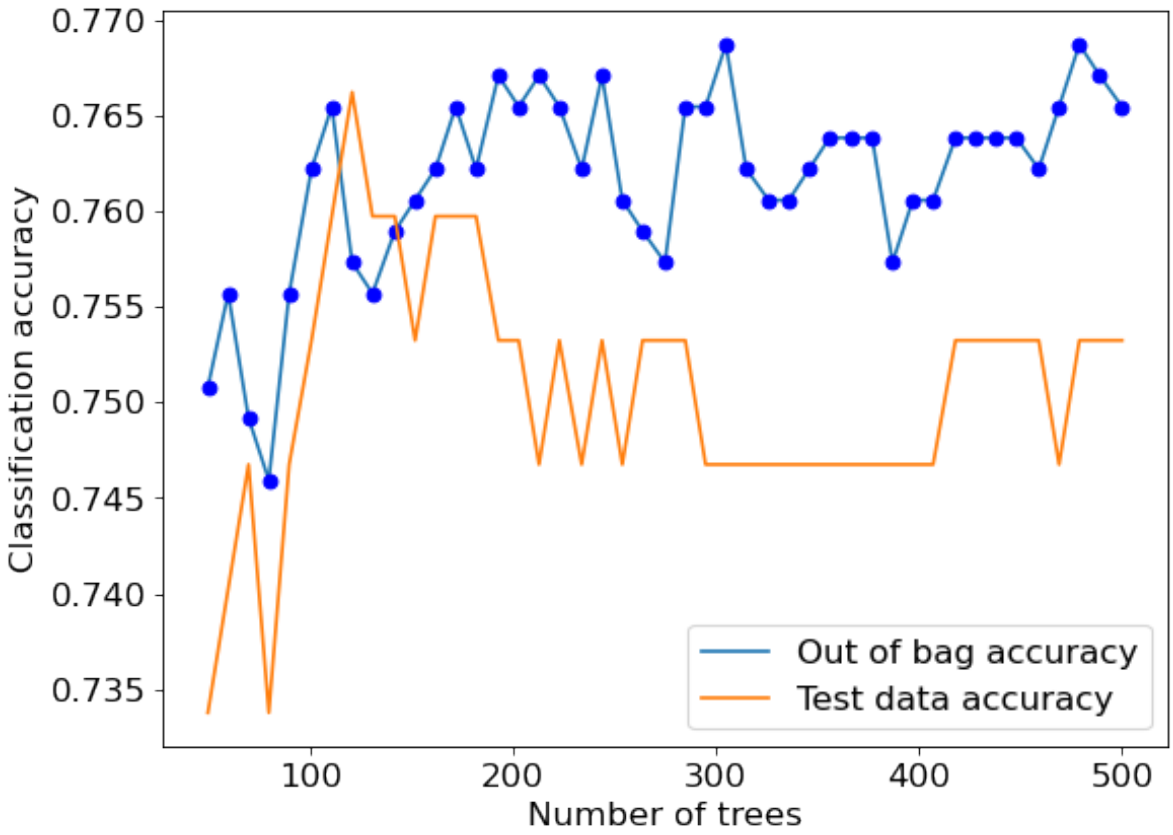
The model obtained above is similar to the one obtained by bagging. We'll discuss the comparison later.

6.2.1 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_jo
    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')
```

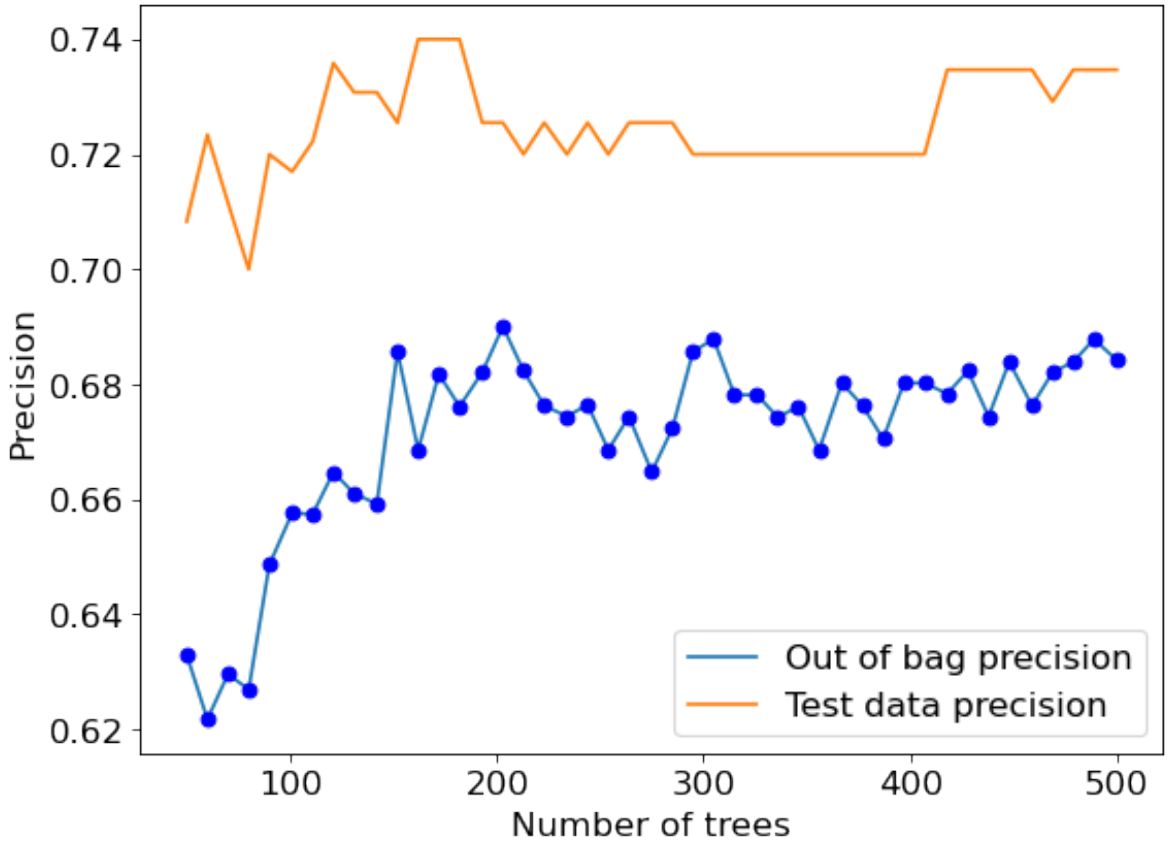
```
plt.plot(test_accuracy.keys(),test_accuracy.values(), label = 'Test data accuracy')

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();
```



6.2.2 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_estimator
                                   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)])
```

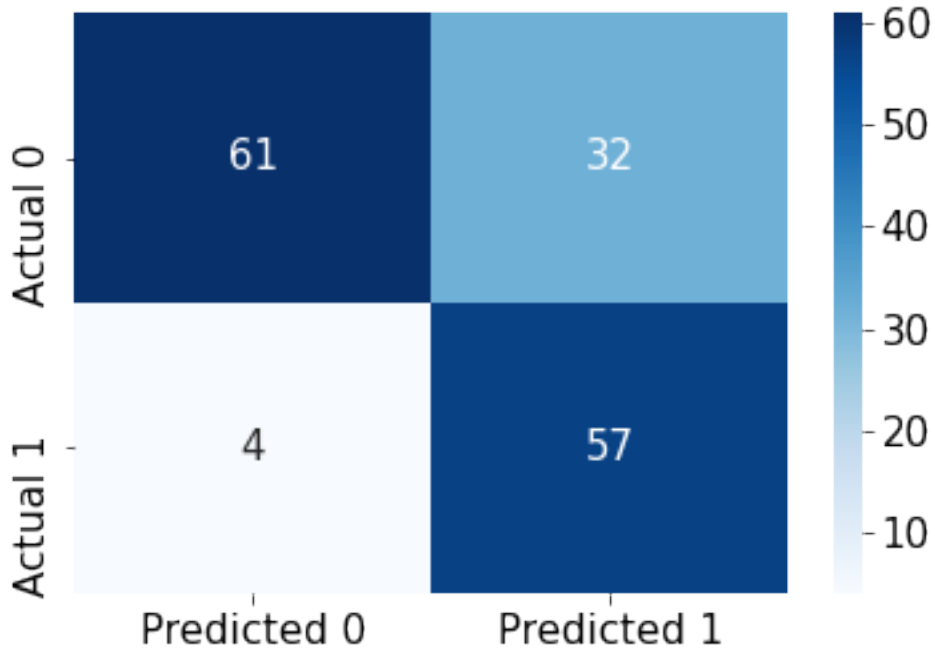
```
time taken =  0.08032723267873128  minutes
max recall =  0.5990338164251208
params=  (500, 8)
```

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

```
# 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 g
# 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 ])
```

6.3 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 variab
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

```
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)
```

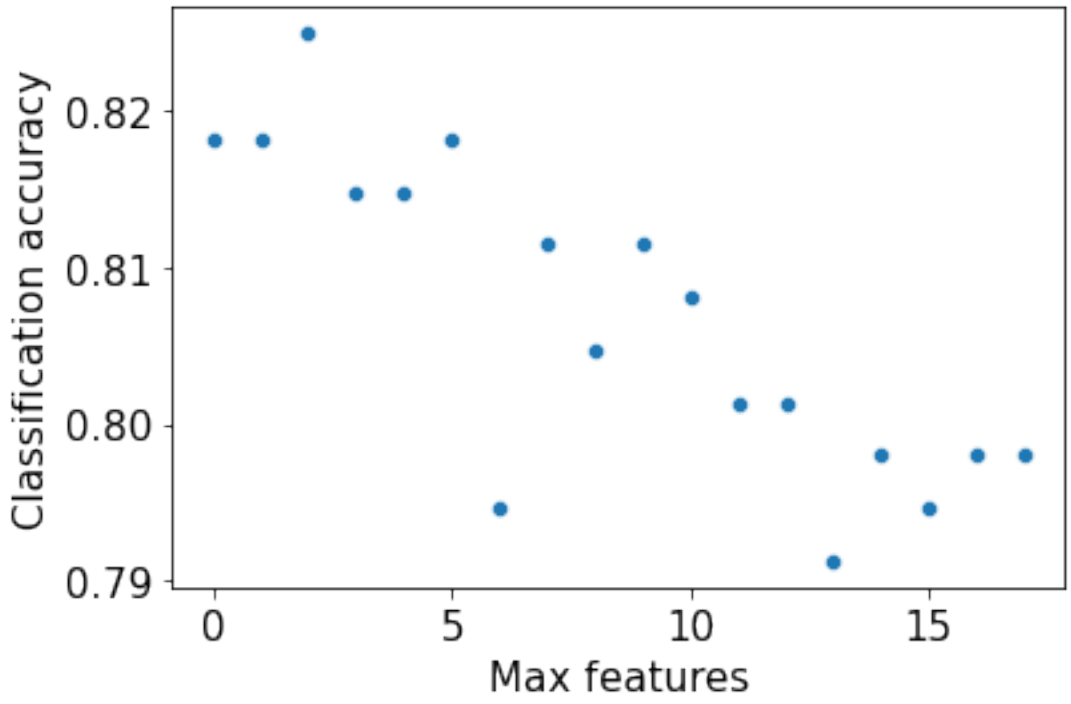
Tuning random forest

```
#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_estimator
                                   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_accuacy2={};

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

```
    #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
    test_accuacy2[i]=model2.score(Xtest,ytest) #Returns the classification accuracy of the
```

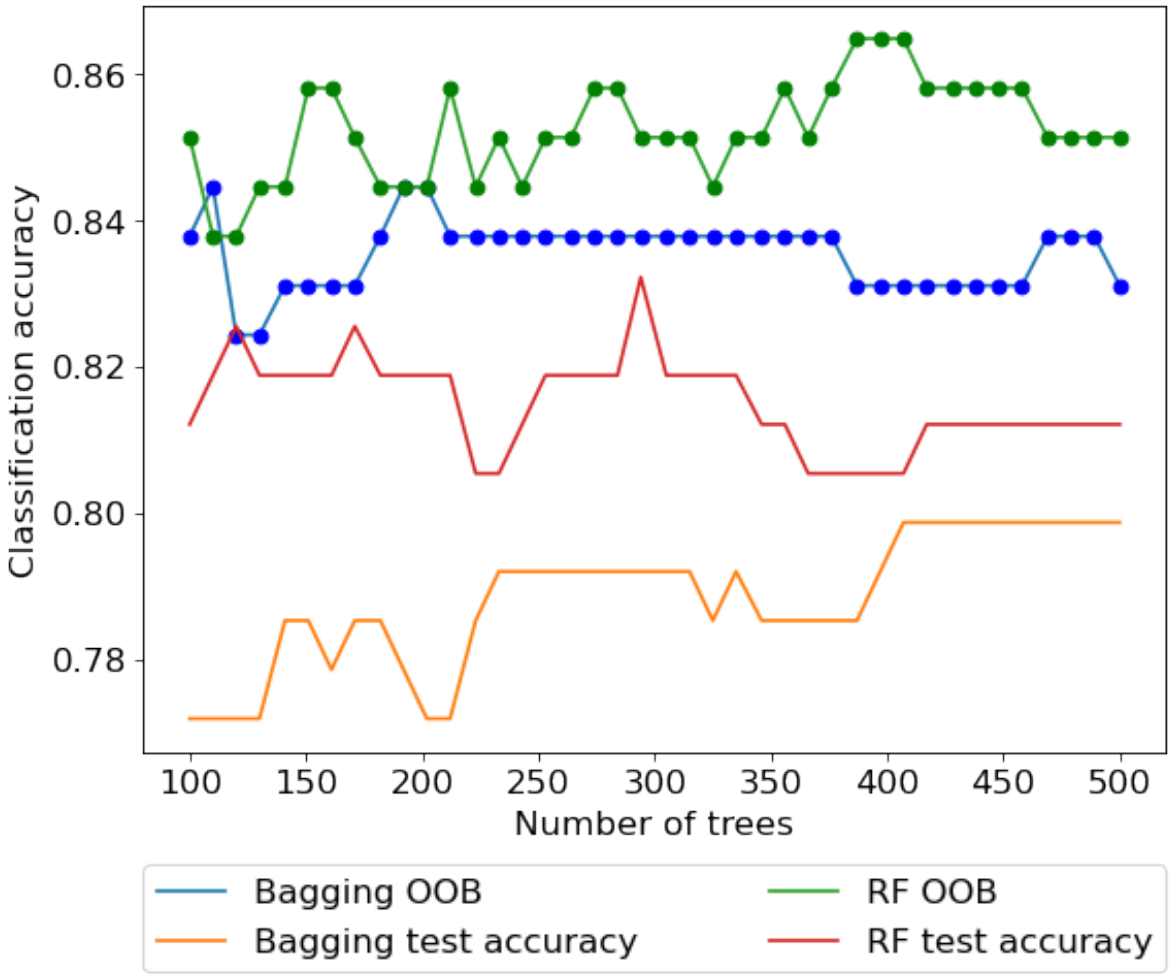
```
#Feature importance for bagging
np.mean([tree.feature_importances_ for tree in model.estimators_],axis=0)
```

```
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_accuacy2.keys(),test_accuacy2.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)
```

```
<matplotlib.legend.Legend at 0x187bd2a2640>
```



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.

7 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.

7.1 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_pre
from sklearn.metrics import mean_squared_error,r2_score,roc_curve,auc,precision_recall_cur
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,AdaBoost
from sklearn.linear_model import LinearRegression,LogisticRegression
from sklearn.neighbors import KNeighborsRegressor
import itertools as it
import time as time
```

```
#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']
```

7.2 AdaBoost for regression

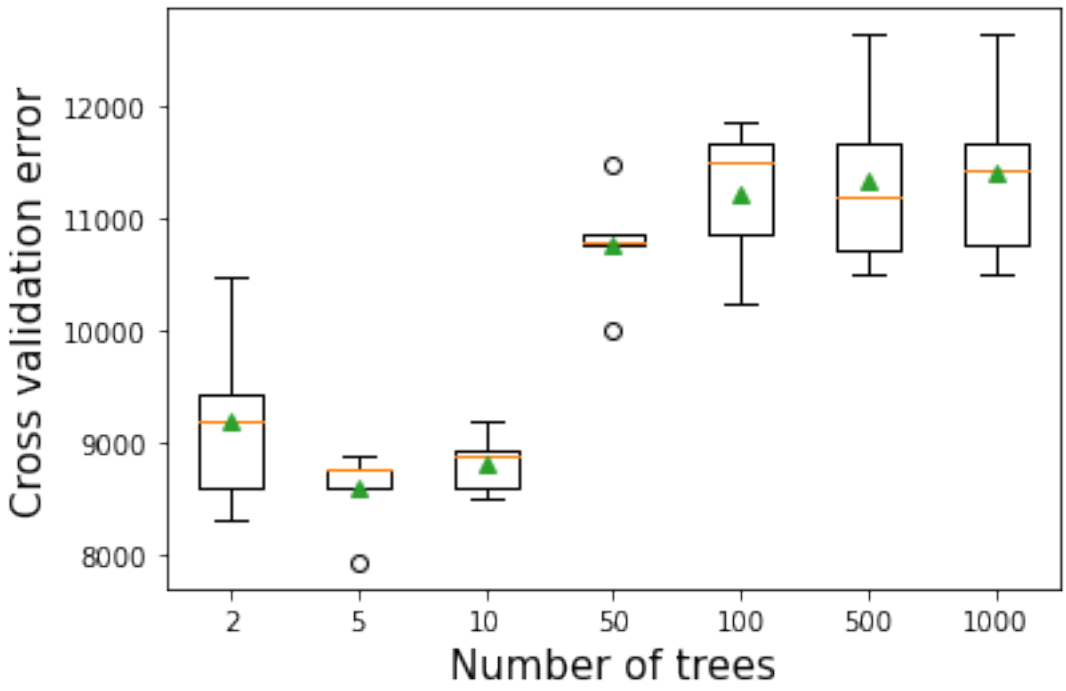
7.2.1 Number of trees vs cross validation error

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 = np.sqrt(-cross_val_score(model, X, y, scoring='neg_mean_squared_error', cv=cv
    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)

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



7.2.2 Depth of tree vs cross validation error

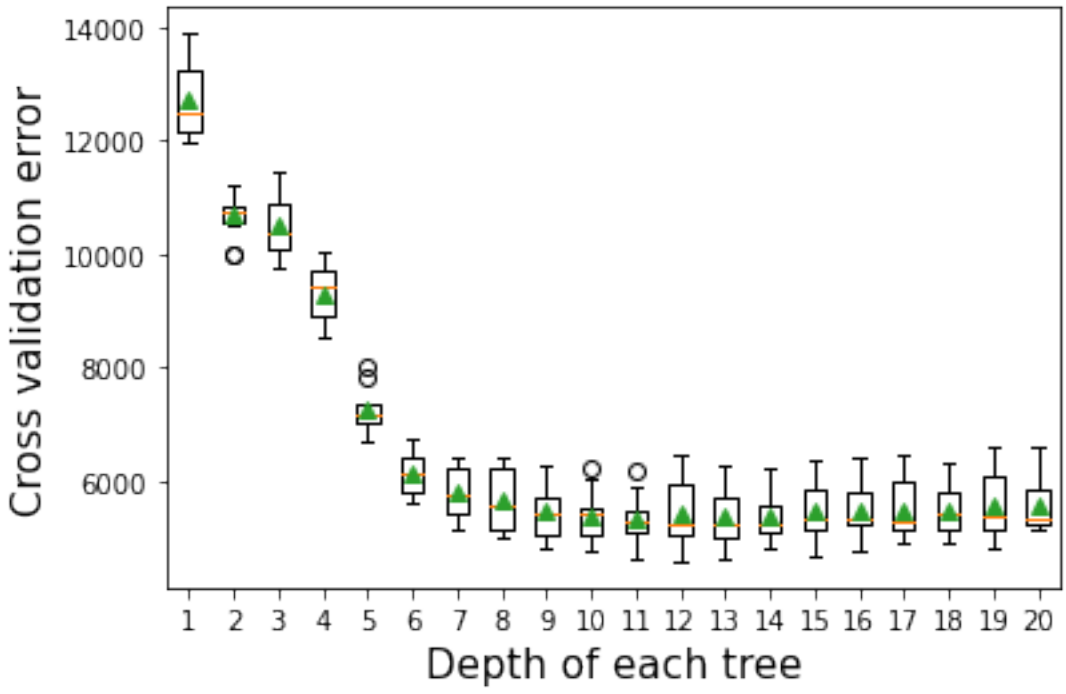
As the depth of each weak learner (decision tree) increases, the complexity of the weak learner will increase. As the complexity increases, the prediction bias will decrease, while the prediction variance will increase. Thus, there will be an optimal depth for each weak learner that minimizes the prediction 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 = np.sqrt(-cross_val_score(model, X, y, scoring='neg_mean_squared_error', cv=cv
    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 12704.191 (661.913)
>2 10675.975 (382.400)
>3 10523.960 (557.974)
>4 9303.664 (500.022)
>5 7257.473 (385.578)
>6 6120.387 (371.625)
>7 5802.894 (428.146)
>8 5656.343 (521.073)
>9 5449.504 (471.809)
>10 5379.424 (452.370)
>11 5330.506 (428.361)
>12 5416.617 (580.948)
>13 5371.431 (495.273)
>14 5368.026 (417.437)
>15 5477.644 (538.878)
>16 5477.425 (468.785)
>17 5495.560 (520.657)
>18 5489.784 (462.329)
>19 5577.452 (564.083)
>20 5563.340 (479.502)
```

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



7.2.3 Learning rate vs cross validation error

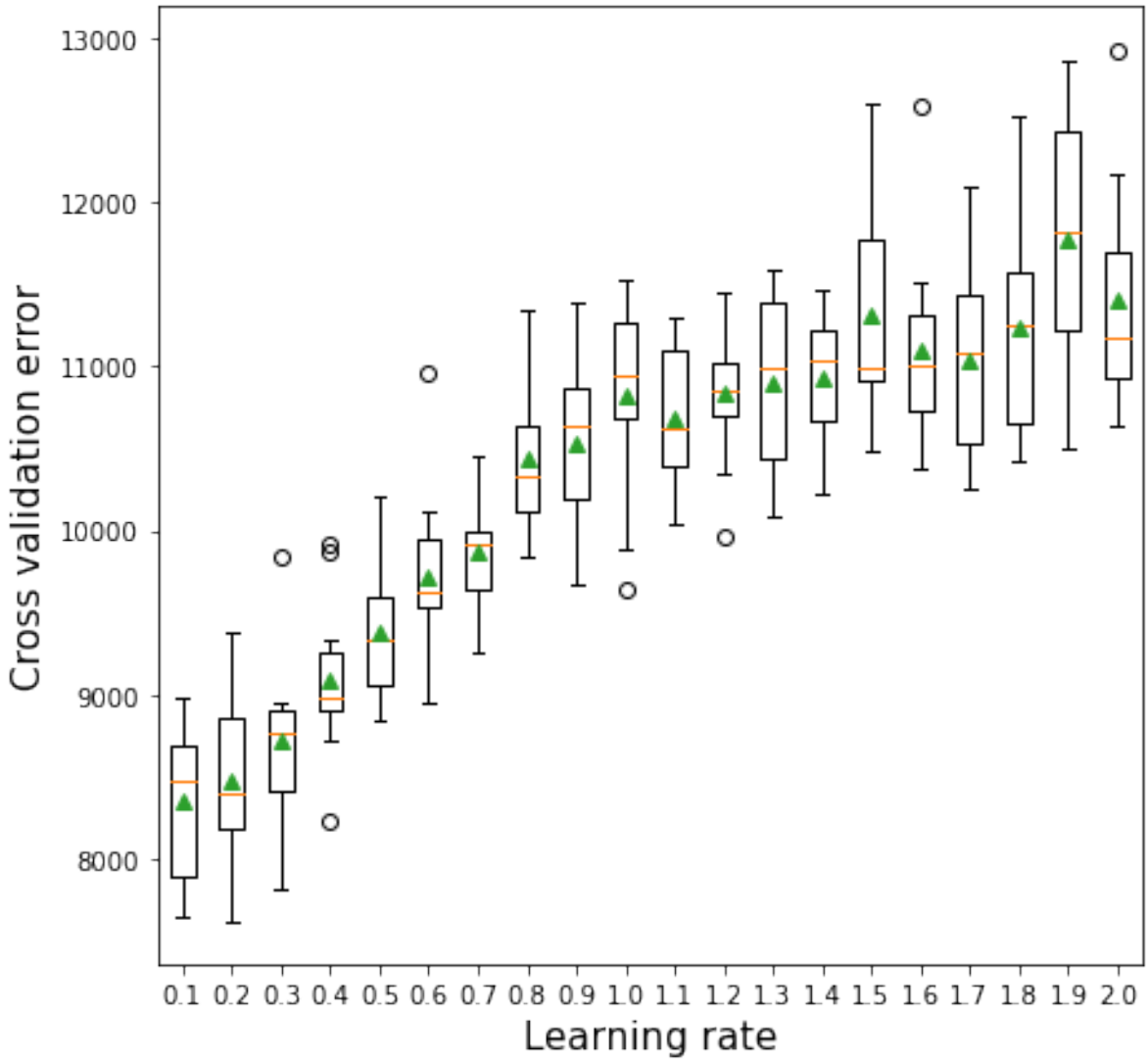
The optimal learning rate will depend on the number of trees, and vice-versa. If the learning rate is too low, it will take several trees to "learn" the response. If the learning rate is high, the response will be "learned" quickly (with fewer) trees. Learning too quickly will be prone to overfitting, 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 = np.sqrt(-cross_val_score(model, X, y, scoring='neg_mean_squared_error', cv=cv
    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 8347.3 (460.1)
>0.2 8478.3 (487.2)
>0.3 8717.4 (500.1)
>0.4 9091.3 (481.2)
>0.5 9374.3 (400.4)
>0.6 9712.9 (548.1)
>0.7 9866.8 (359.5)
>0.8 10443.6 (436.4)
>0.9 10526.6 (503.4)
>1.0 10821.8 (594.8)
>1.1 10688.5 (437.6)
>1.2 10840.2 (426.6)
>1.3 10892.1 (543.8)
>1.4 10932.6 (388.1)
>1.5 11316.6 (656.2)
>1.6 11098.3 (596.2)
>1.7 11037.3 (560.1)
>1.8 11236.2 (650.7)
```

```
>1.9 11770.4 (749.1)
>2.0 11404.9 (681.6)

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



7.2.4 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['base_estimator'] = [DecisionTreeRegressor(max_depth=3), DecisionTreeRegressor(max_de
                          DecisionTreeRegressor(max_depth=10),DecisionTreeRegressor(max_de
# 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='ne
# 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: -28598146.516266 using {'base_estimator': DecisionTreeRegressor(max_depth=10), 'learni
```

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 base estimator by adding a double underscore "`__`" between the `base_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 base estimator within the `AdaBoostRegressor()` function.

```
model = AdaBoostRegressor(random_state=1, base_estimator = DecisionTreeRegressor(random_st
grid = dict()
grid['n_estimators'] = [10, 50, 100,200]
grid['learning_rate'] = [0.0001, 0.001, 0.01,0.1, 1.0]
grid['base_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='ne
# 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: -28598146.516266 using {'base_estimator__max_depth': 10, 'learning_rate': 1.0, 'n_estim
```

```
#Model based on the optimal hyperparameters
model = AdaBoostRegressor(base_estimator=DecisionTreeRegressor(max_depth=10),n_estimators=
                          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
```

```
model2 = RandomForestRegressor(n_estimators=300, random_state=1,
                          n_jobs=-1, max_features=2).fit(X, y)
pred2 = model2.predict(Xtest)
print("Random Forest model RMSE = ", np.sqrt(mean_squared_error(model2.predict(Xtest),ytes
```

```
Random Forest model RMSE =  5642.45839697972
```

```
#Ensemble modeling
pred = 0.5*pred1+0.5*pred2
print("Ensemble model RMSE = ", np.sqrt(mean_squared_error(pred,ytest)))
```

```
Ensemble model RMSE =  5528.699297204213
```

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

7.3 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']
```

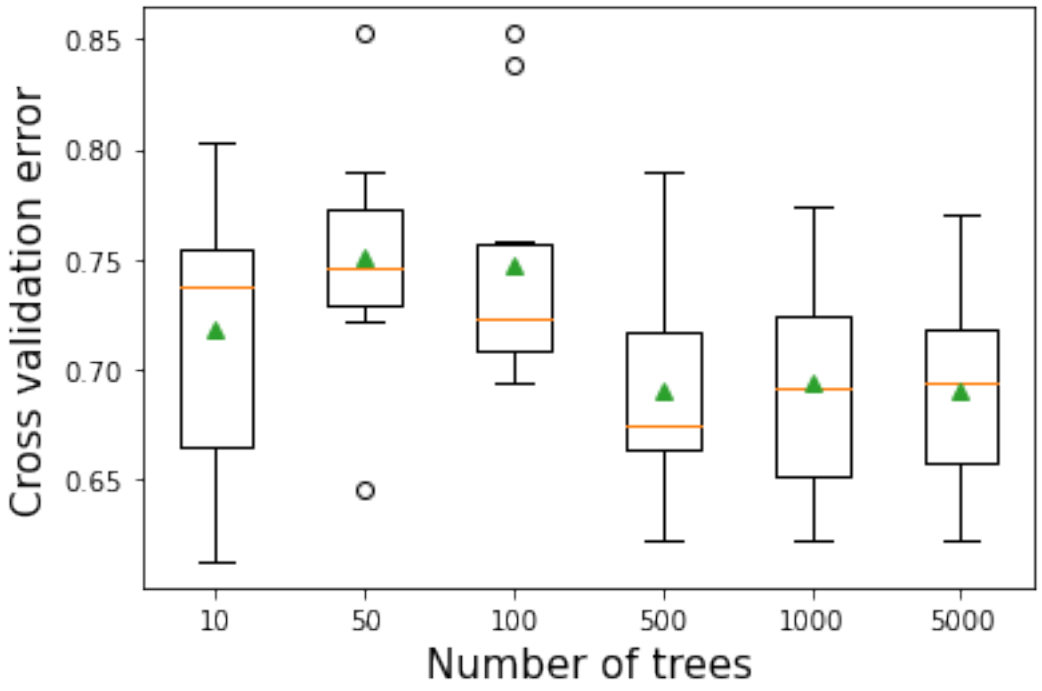
7.3.1 Number of trees vs cross validation accuracy

```
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')
```



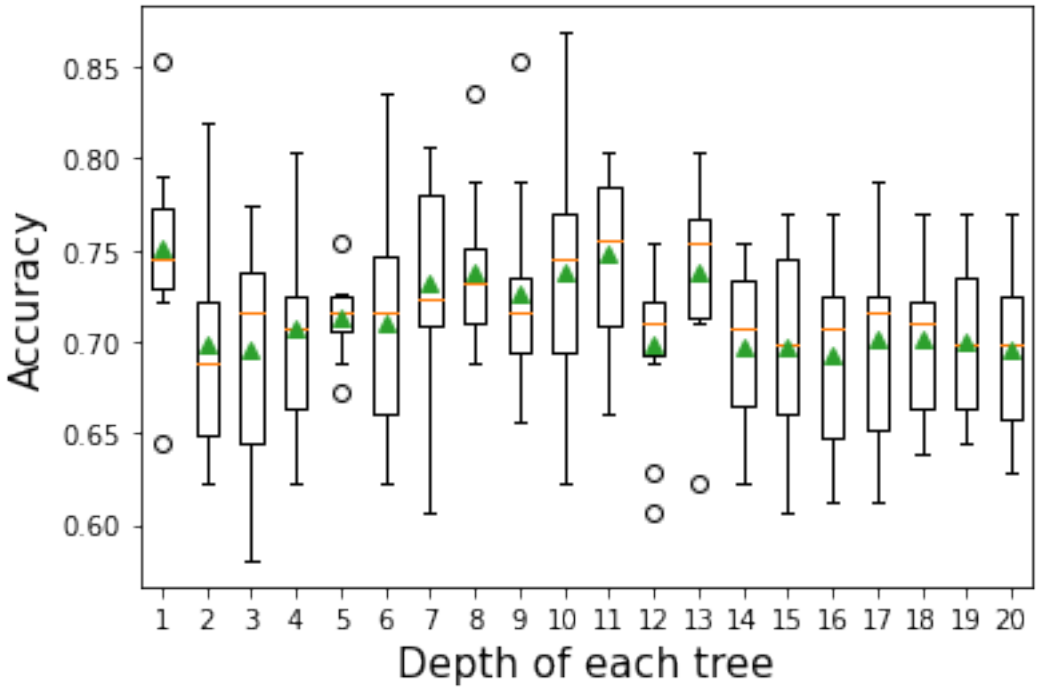
7.3.2 Depth of each tree vs cross validation accuracy

```
# 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(base_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')
```

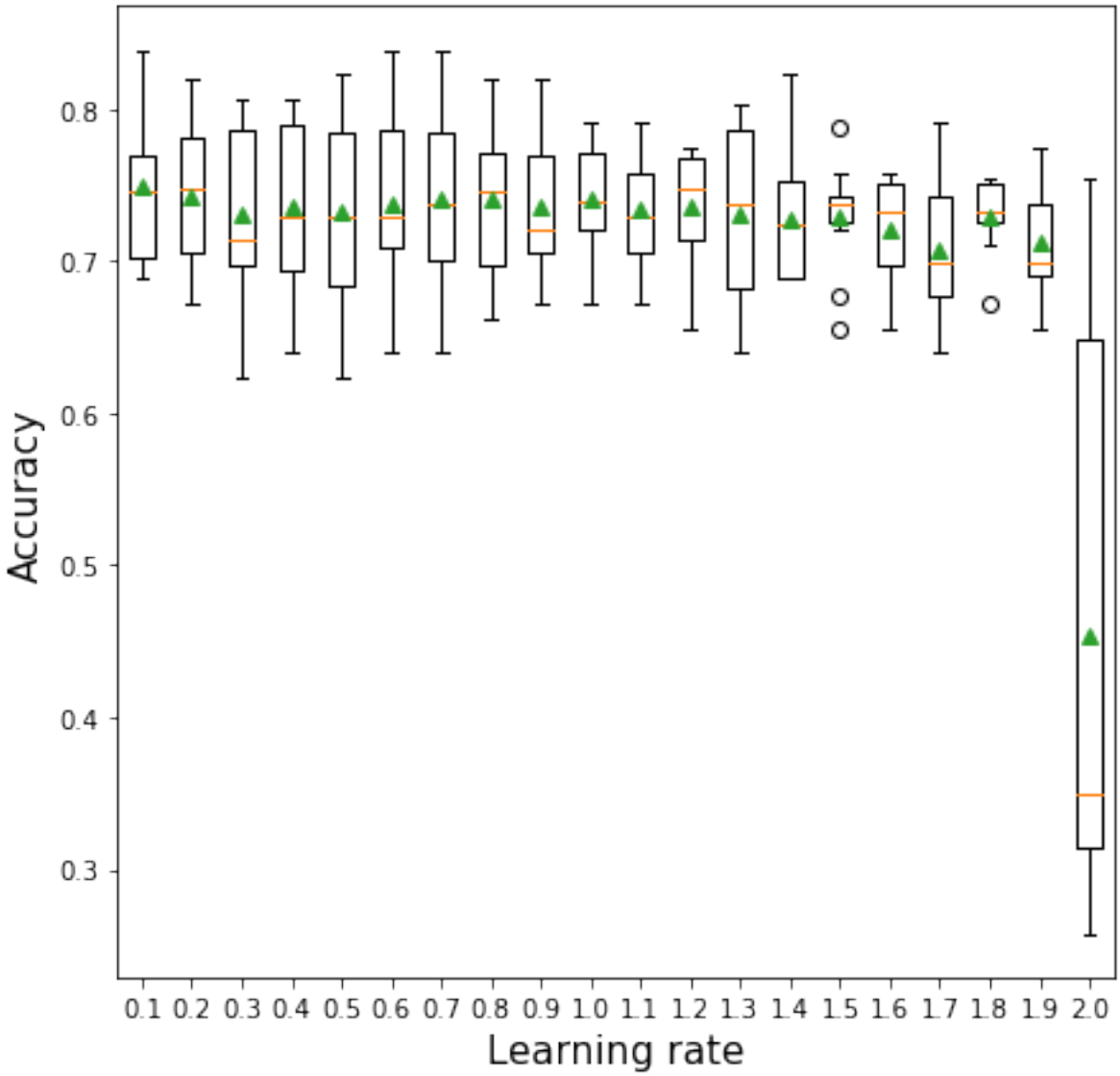


7.3.3 Learning rate vs cross validation accuracy

```
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')
```



7.3.4 Tuning AdaBoost Classifier hyperparameters

```
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_
                          DecisionTreeClassifier(max_depth=3),DecisionTreeClassifier(max_d
```

```
# 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)
```

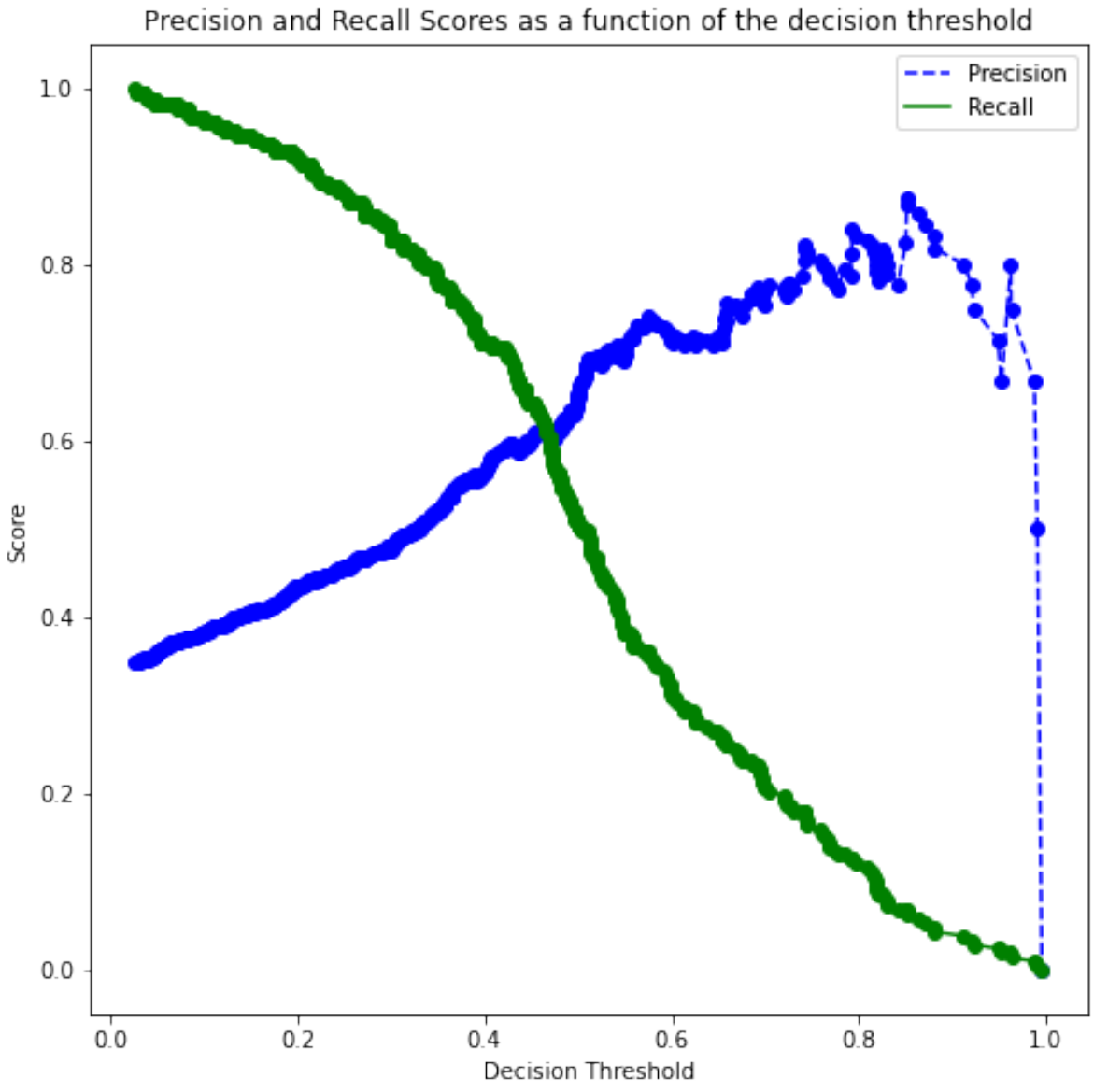
```
Fitting 5 folds for each of 100 candidates, totalling 500 fits
Best: 0.763934 using {'base_estimator': DecisionTreeClassifier(max_depth=3), 'learning_rate'
```

7.3.5 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,base_estimator = DecisionTreeClassifier(max_dept
                           n_estimators=200).fit(X,y)

# Note that we are using the cross-validated predicted probabilities, instead of directly
# predicted probabilities on train data, as the model may be overfitting on the train data
# may lead to misleading results
cross_val_ypred = cross_val_predict(AdaBoostClassifier(random_state=1,base_estimator = Dec
                           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].re
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 in
# the last value will provide the maximum threshold probability for the recall to be more
# We wish to find the maximum threshold probability to obtain the maximum possible precisi
recall_more_than_80[recall_more_than_80.shape[0]-1]
```

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

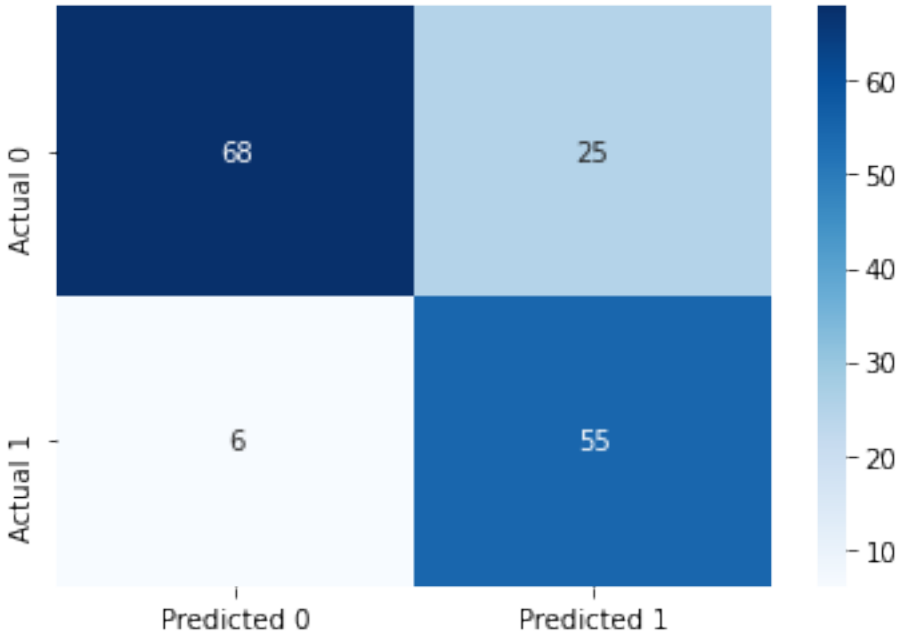
```
#Optimal decision threshold probability
thres = recall_more_than_80[recall_more_than_80.shape[0]-1][0]
thres
```

```
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 g
# 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
ROC-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.

8 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.

8.1 Hyperparameters

There are 4 important parameters to tune in Gradient boosting:

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

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_pre
from sklearn.metrics import mean_squared_error,r2_score,roc_curve,auc,precision_recall_cur
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, Bagging
from sklearn.linear_model import LinearRegression,LogisticRegression
from sklearn.neighbors import KNeighborsRegressor
import itertools as it
```

```
import time as time
```

```
#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']
```

8.2 Gradient boosting for regression

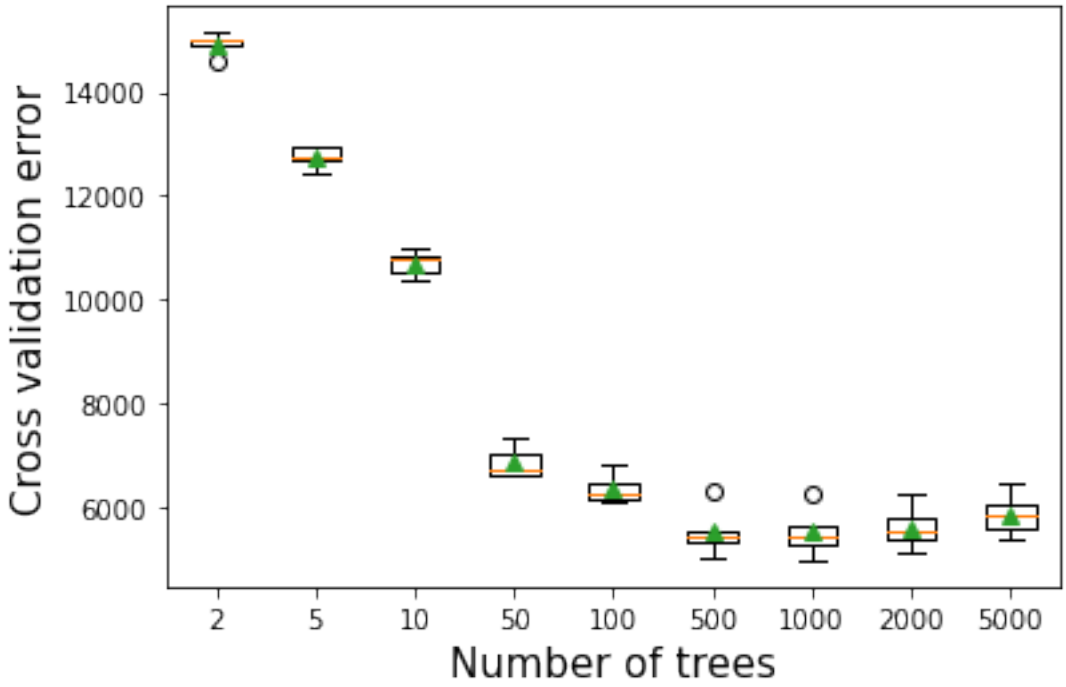
8.2.1 Number of trees vs cross validation error

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

```
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='hub
```

```
    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
    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')
```



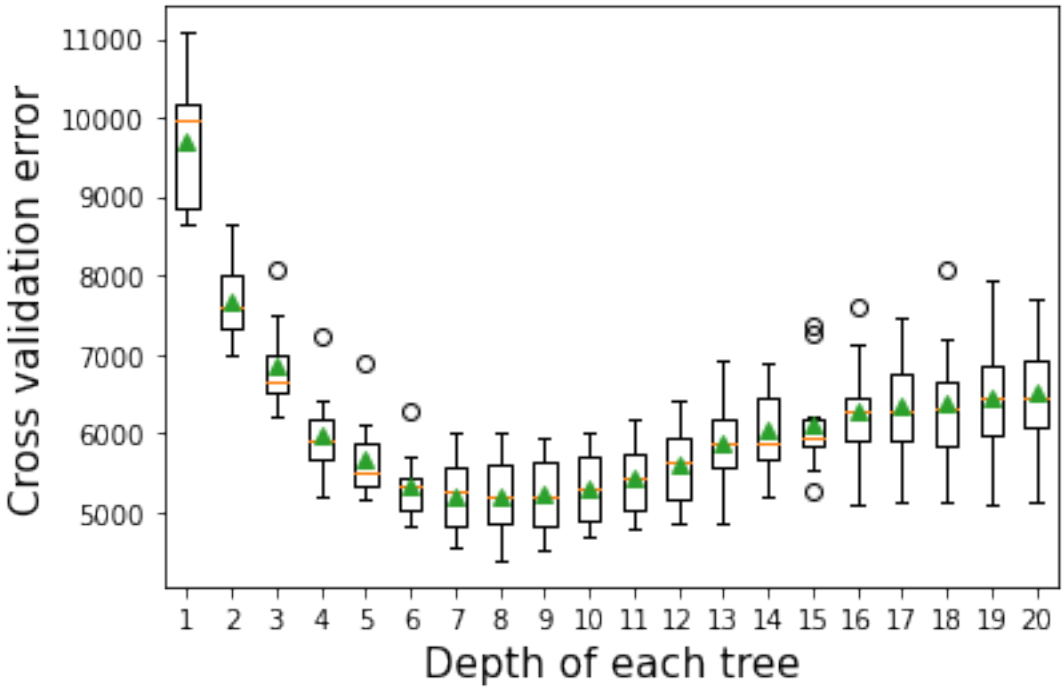
8.2.2 Depth of tree vs cross validation error

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_dept
    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
    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')
```



8.2.3 Learning rate vs cross validation error

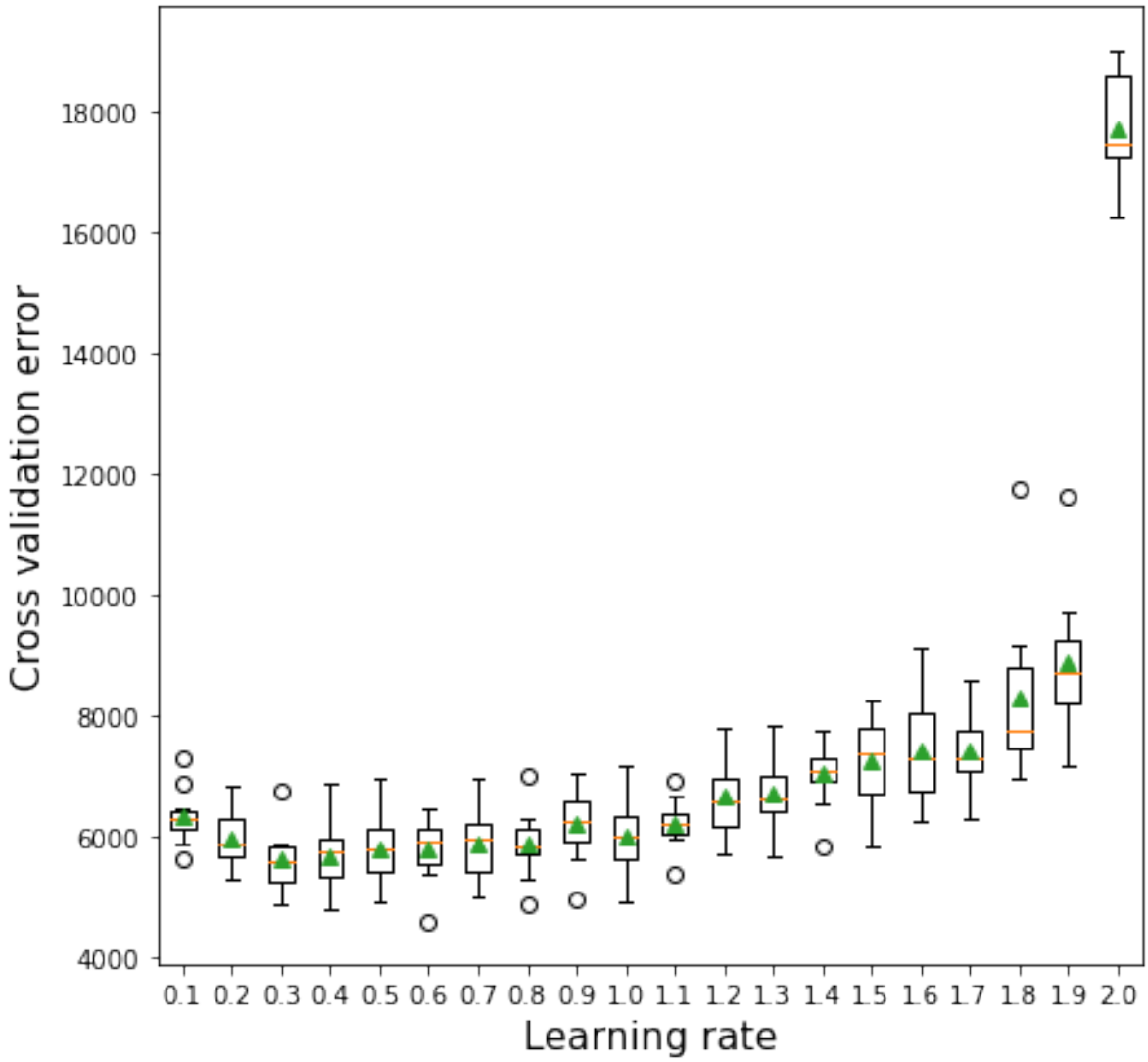
The optimal learning rate will depend on the number of trees, and vice-versa. If the learning rate is too low, it will take several trees to "learn" the response. If the learning rate is high, the response will be "learned" quickly (with fewer) trees. Learning too quickly will be prone to overfitting, 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
    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')
```



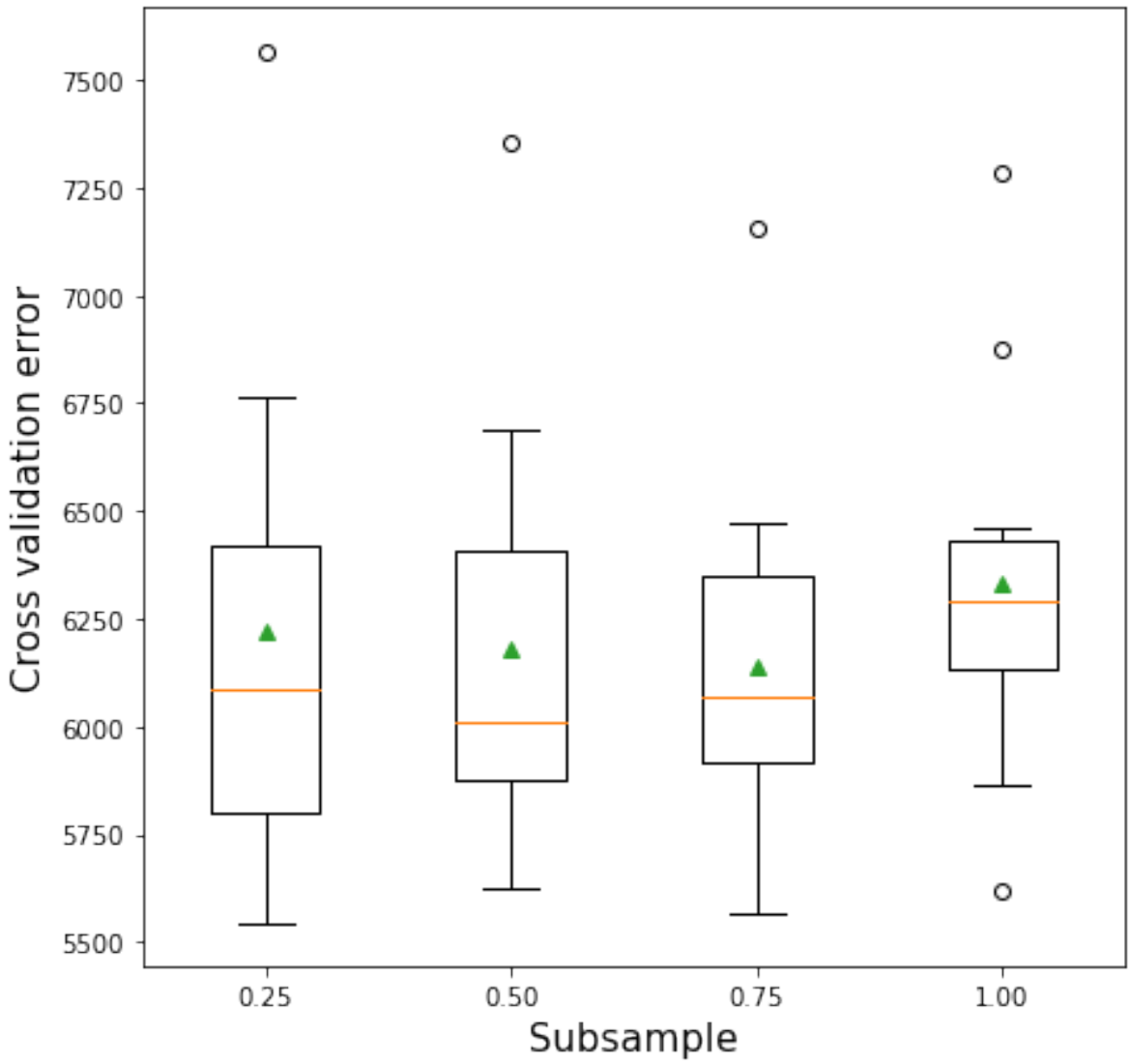
8.2.4 Subsampling vs cross validation error

```
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
        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')
```



8.2.5 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='ne
                           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 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
```

```
#Let us combine the Gradient boost model with other models
model2 = AdaBoostRegressor(base_estimator=DecisionTreeRegressor(max_depth=10),n_estimators
                          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
```

8.2.6 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.

8.3 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']
```

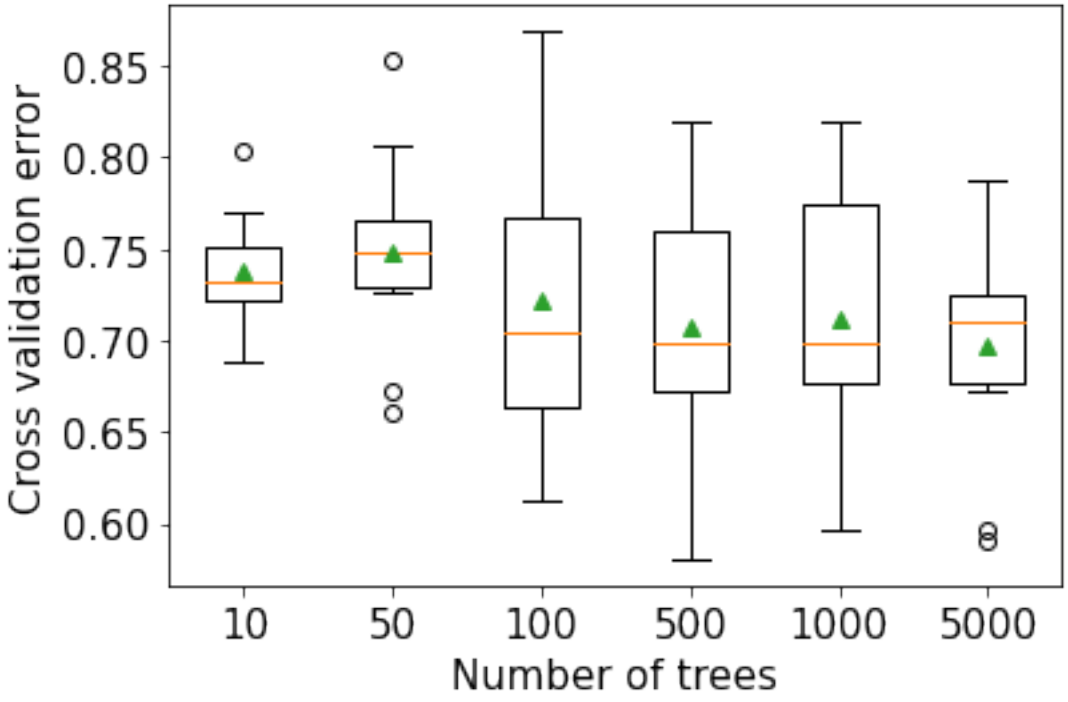
8.3.1 Number of trees vs cross validation accuracy

```
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')
```



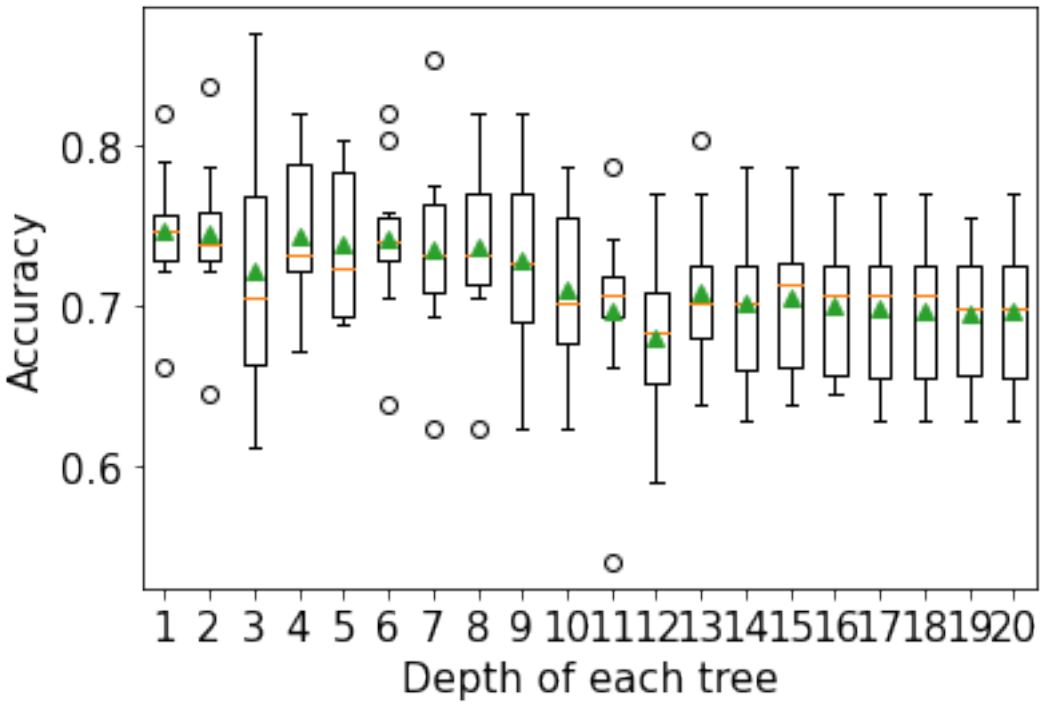
8.3.2 Depth of each tree vs cross validation accuracy

```
# 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')
```

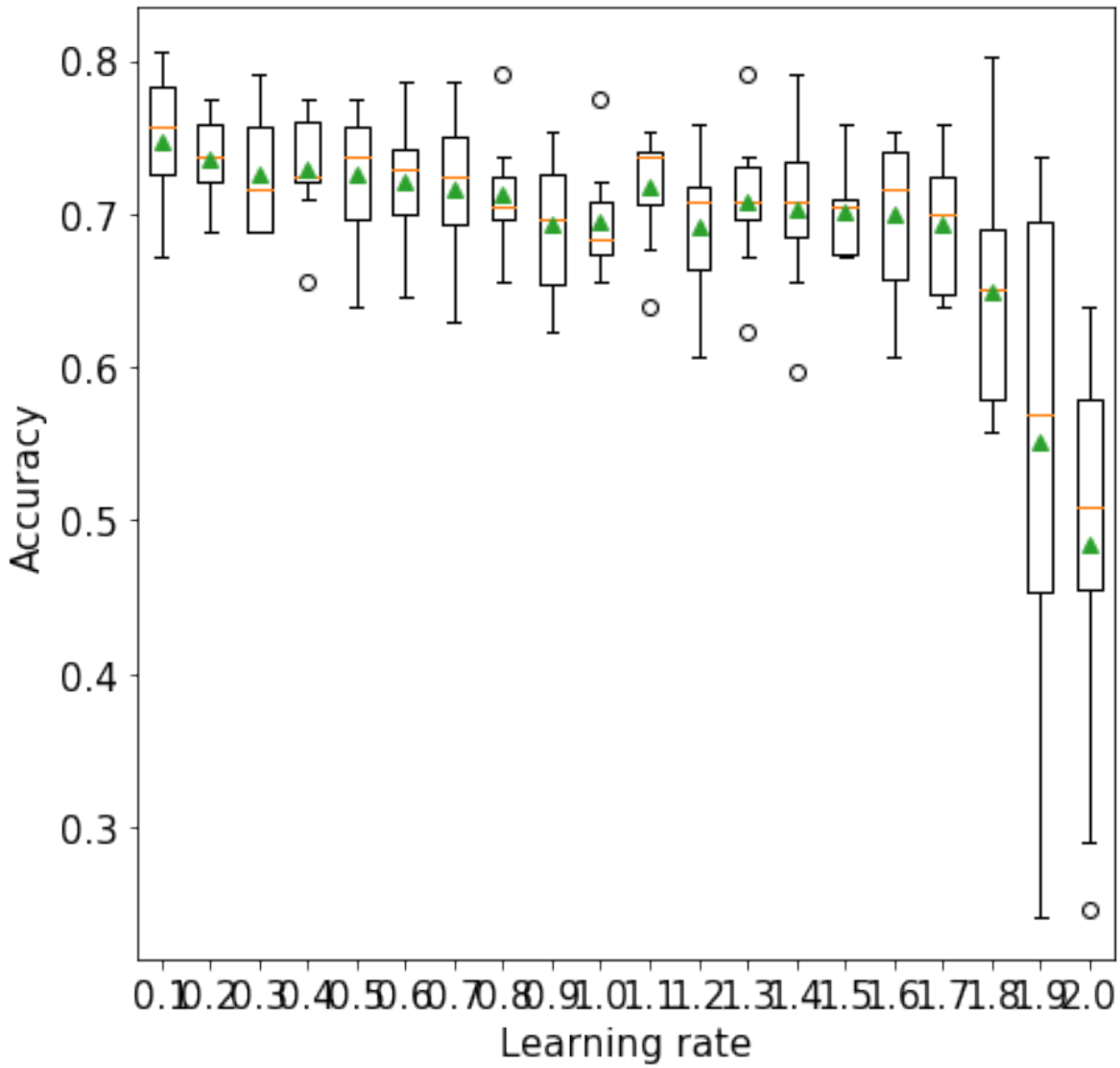


8.3.3 Learning rate vs cross validation accuracy

```
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')
```



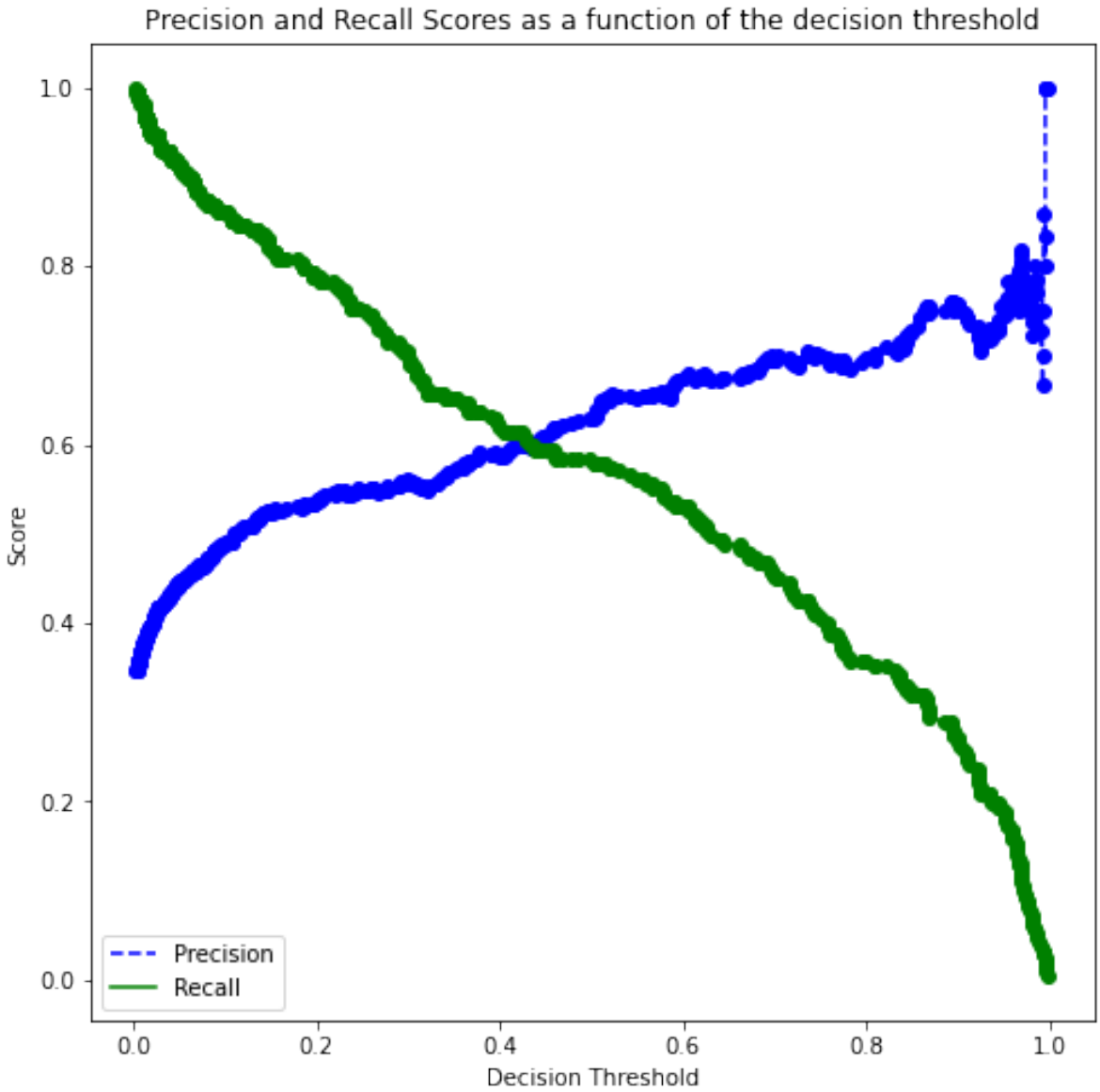
8.3.4 Tuning Gradient boosting Classifier

```
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 = T
# 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'
Time taken =  32.46394085884094
```

```
#Model based on the optimal parameters
model = GradientBoostingClassifier(random_state=1,max_depth=3,learning_rate=0.1,subsample=
                           n_estimators=200).fit(X,y)

# Note that we are using the cross-validated predicted probabilities, instead of directly
# predicted probabilities on train data, as the model may be overfitting on the train data
# may lead to misleading results
cross_val_ypred = cross_val_predict(GradientBoostingClassifier(random_state=1,max_depth=3,
                                                               learning_rate=0.1,subsample
                           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].re
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 in
# the last value will provide the maximum threshold probability for the recall to be more
# We wish to find the maximum threshold probability to obtain the maximum possible precisi
recall_more_than_80[recall_more_than_80.shape[0]-1]
```

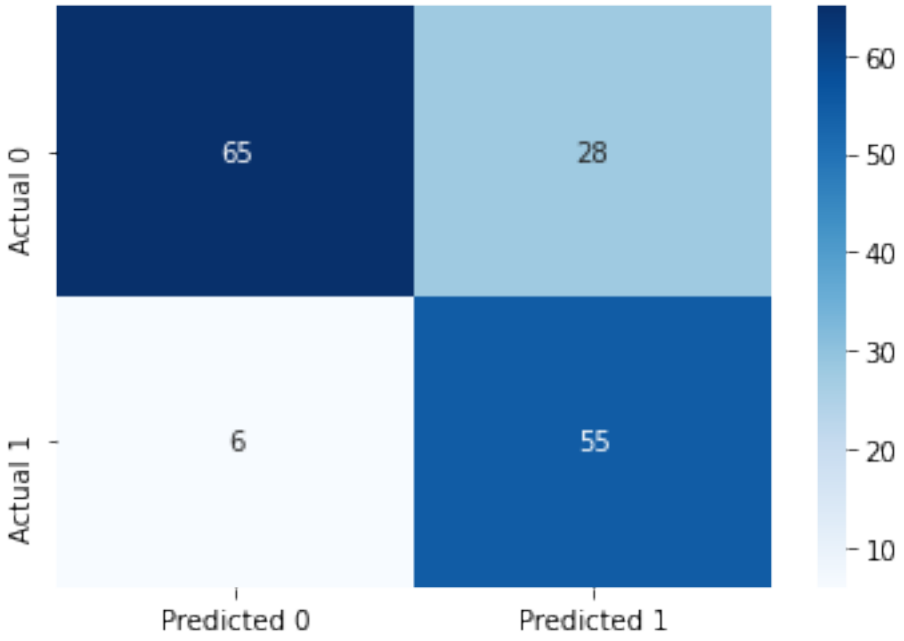
```
array([0.18497144, 0.53205128, 0.80193237])
```

```
#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 g
# 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.

9 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

9.1 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_pre
from sklearn.metrics import mean_squared_error,r2_score,roc_curve,auc,precision_recall_cur
```

```
recall_score, precision_score, confusion_matrix
from sklearn.tree import DecisionTreeRegressor,DecisionTreeClassifier
from sklearn.model_selection import GridSearchCV, ParameterGrid, StratifiedKFold, Randomiz
from sklearn.ensemble import VotingRegressor, VotingClassifier, StackingRegressor, Stackin
from sklearn.linear_model import LinearRegression,LogisticRegression, LassoCV, RidgeCV, El
from sklearn.neighbors import KNeighborsRegressor
import itertools as it
import time as time
import xgboost as xgb
from pyearth import Earth
```

```
#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 XGBoost for regression

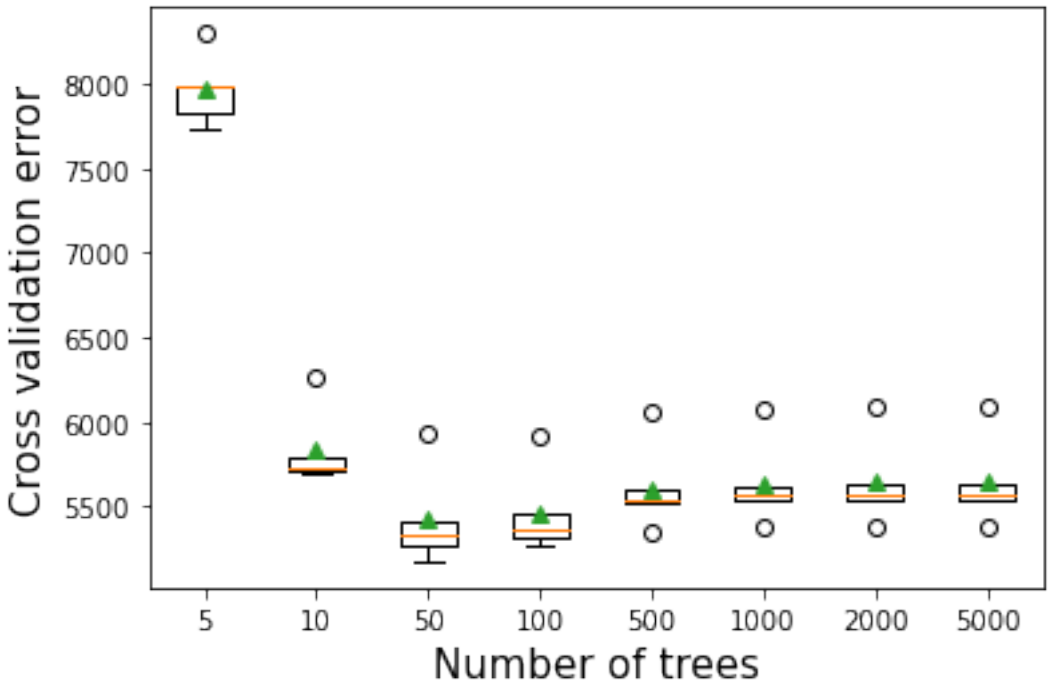
9.2.1 Number of trees vs cross validation error

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

```
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
    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')
```



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 of each weak learner that minimizes the prediction error.

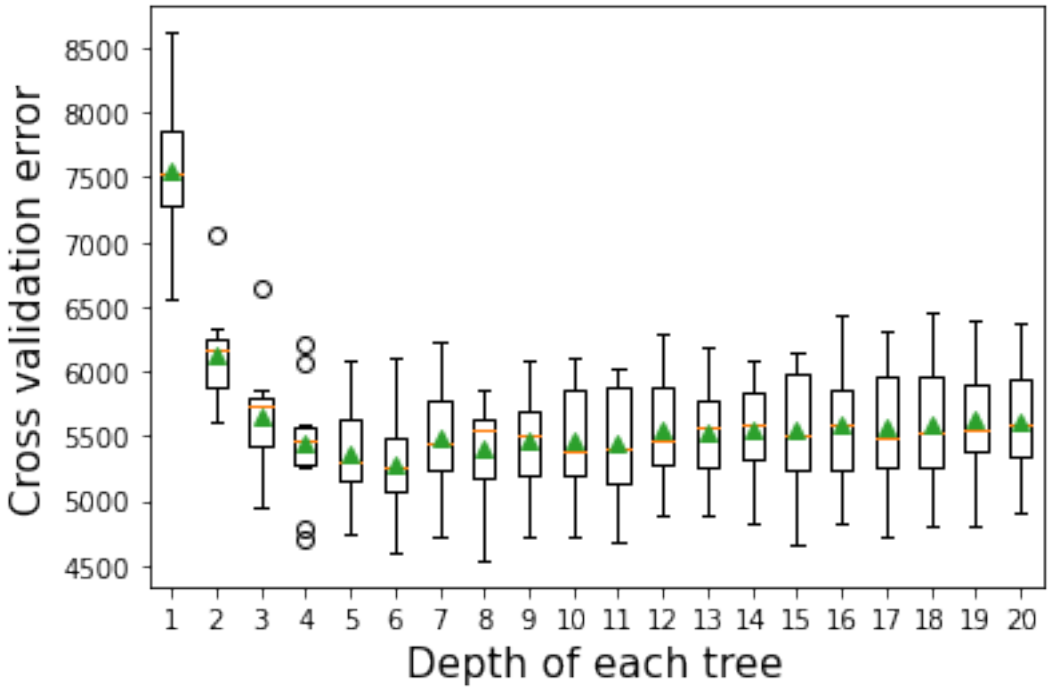
```
# get a list of models to evaluate
def get_models():
    models = dict()
```

```
    # 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
    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')
```



9.2.3 Learning rate vs cross validation error

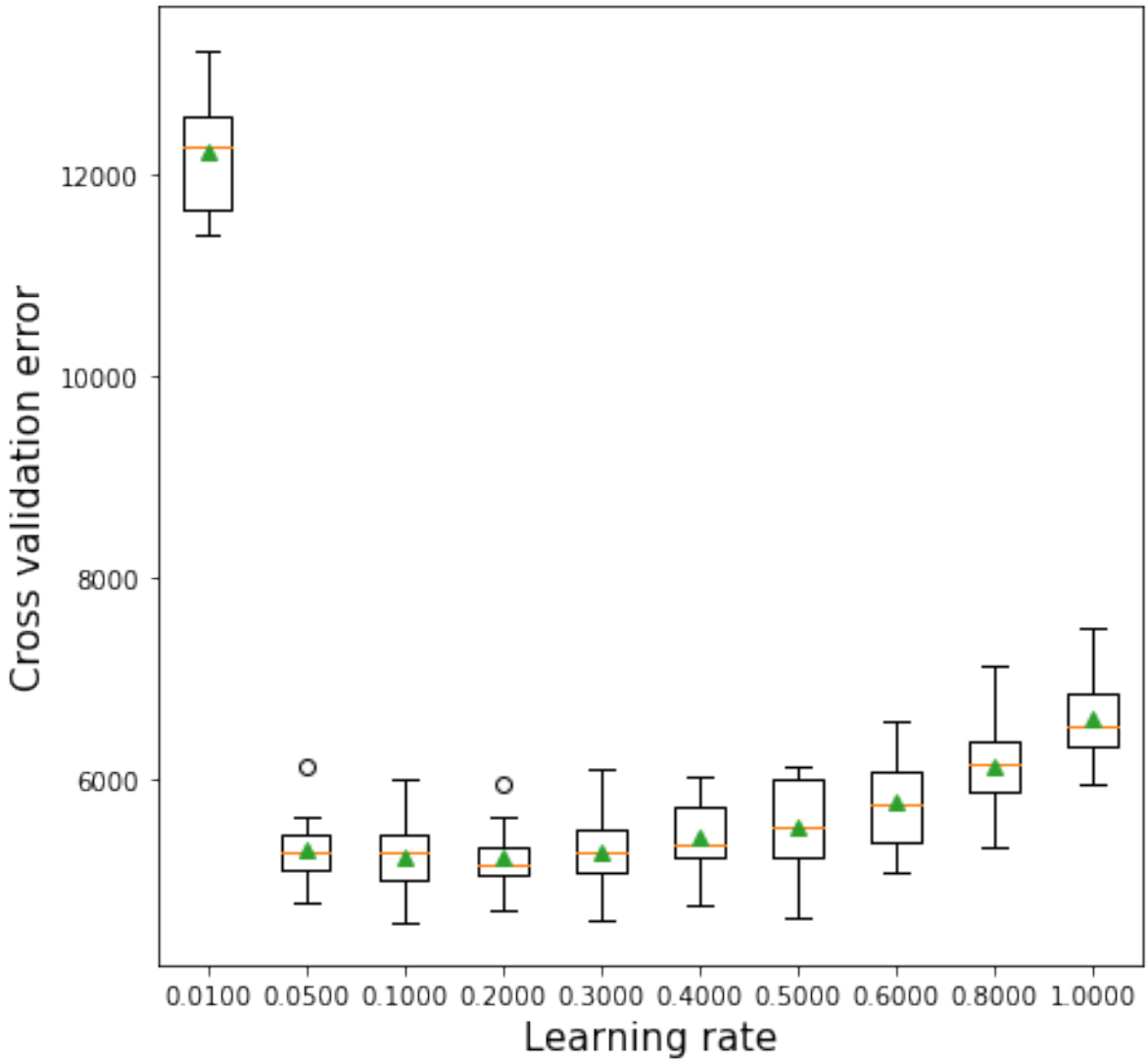
The optimal learning rate will depend on the number of trees, and vice-versa. If the learning rate is too low, it will take several trees to "learn" the response. If the learning rate is high, the response will be "learned" quickly (with fewer) trees. Learning too quickly will be prone to overfitting, 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 [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
    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')
```



9.2.4 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.

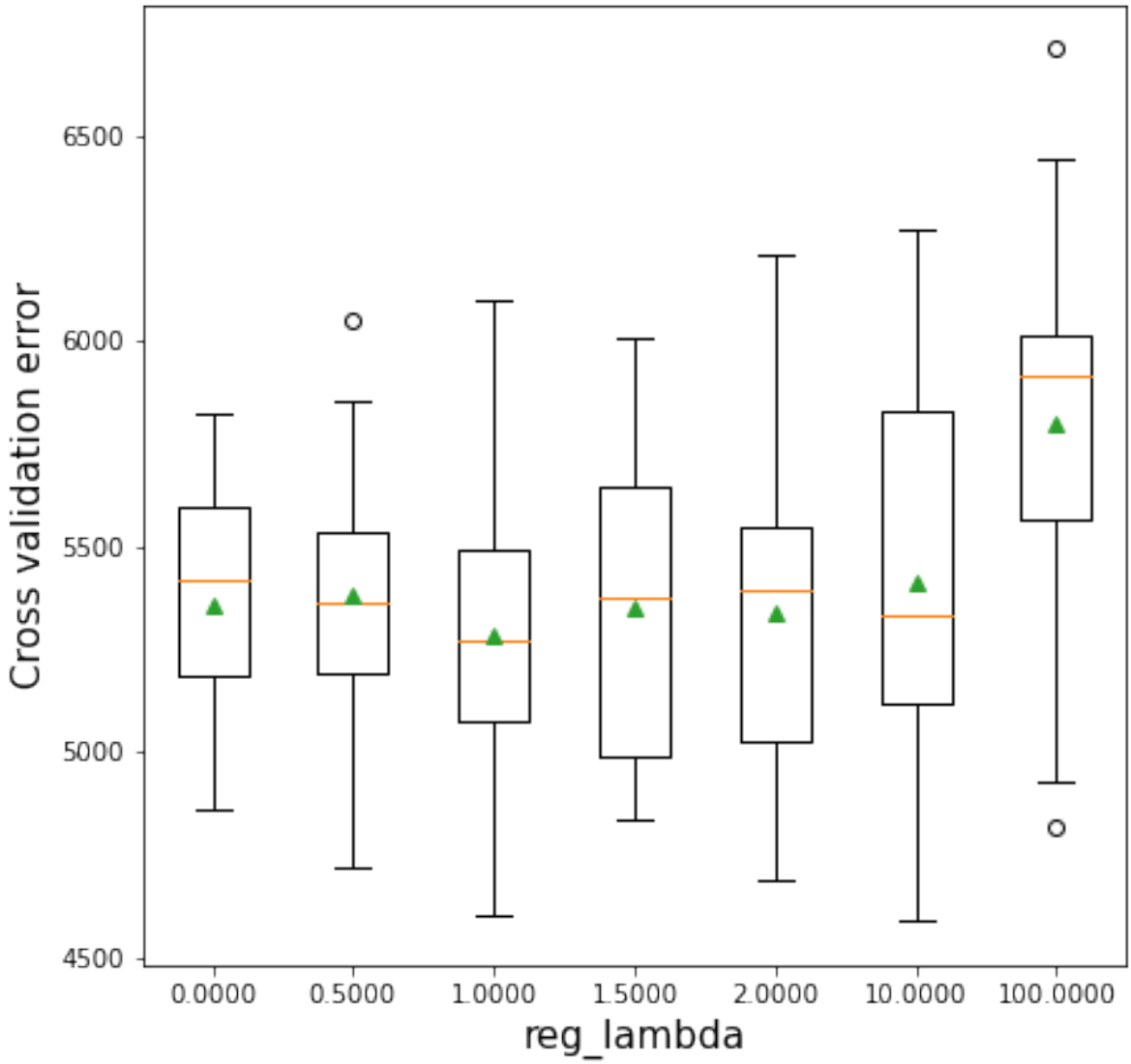
Without regularization, the algorithm will be closer to the gradient boosting algorithm. Regularization may provide some additional boost to prediction accuracy by reducing over-fitting. In the example below, regularization with ***reg_lambda=1*** *turns out to be better than no regularization* (`reg_lambda=0`)*. 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
    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')
```



9.2.5 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.

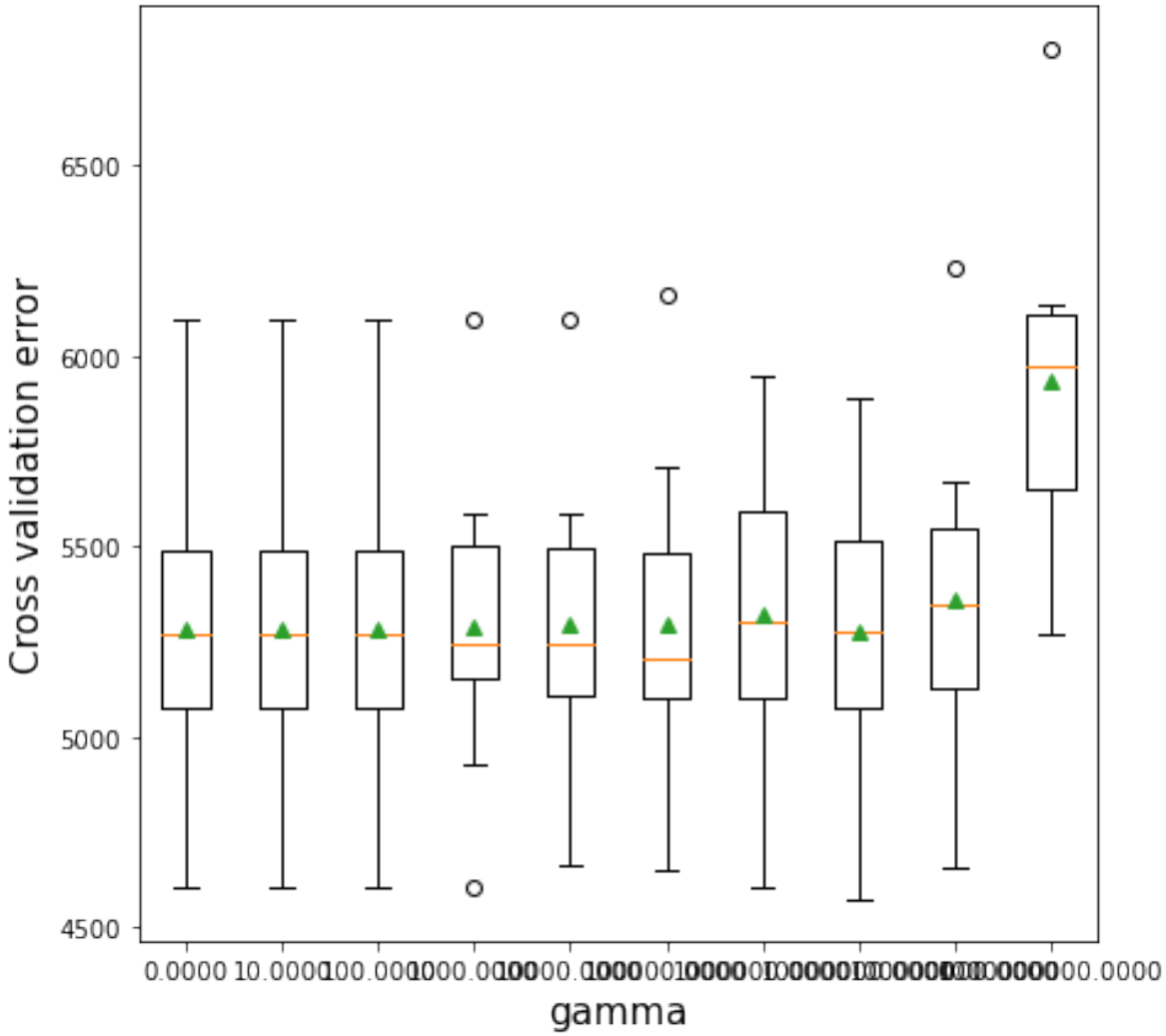
However, in the example below, no regularization (in terms of `gamma=0`) turns out to be better than a non-zero regularization. *(`reg_lambda=0`)*.

```
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
    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)
```

```
>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 5362.1 (407.8)
>1000000000.0000 5932.3 (397.6)
```

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



9.2.6 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_de
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
```

9.2.7 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_stoppin
```

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 10 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_stoppin
```

```
<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_estimat
                        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
#pred = 0.15*pred1+0.15*pred2+0.15*pred3+0.55*pred4 #Option 2 - Higher weight to the bette
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.

9.3 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 varibles replacing the categorical variabl
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 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.

Ref:https://machinelearningmastery.com/xgboost-for-imbalanced-classification/#:~:text=The%20scale_pos_w

```
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 neg
             }

cv = StratifiedKFold(n_splits=5,shuffle=True,random_state=1)
optimal_params = GridSearchCV(estimator=xgb.XGBClassifier(objective = 'binary:logistic',ra
                                                          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_learnin
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
269	1.394361	0.087307	0.005530	0.001718	0.25	0.01

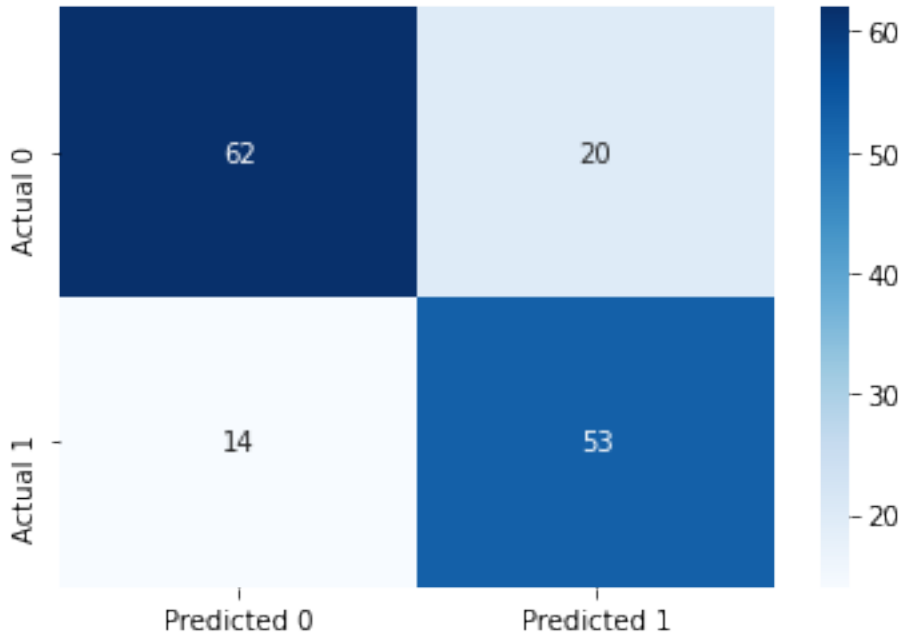
```
#Function to compute confusion matrix and prediction accuracy on test/train data
def confusion_matrix_data(data,actual_values,model,cutoff=0.5):
#Predict the values using the Logit model
    pred_values = model.predict_proba(data)[:,1]
# Specify the bins
    bins=np.array([0,cutoff,1])
#Confusion 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,learnin
                           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
```



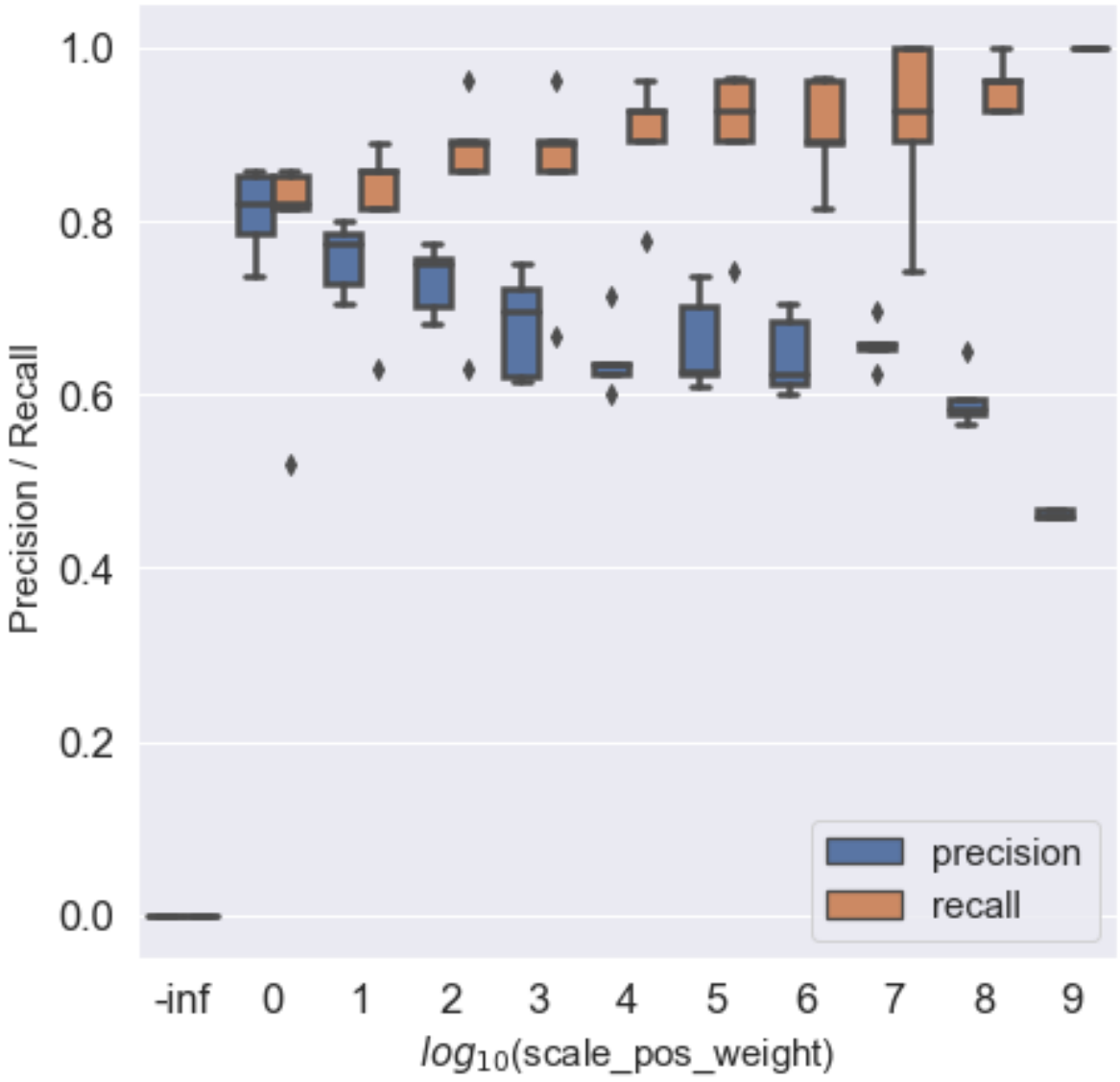
If we increase the value of `scale_pos_weight`, the model will focus on classifying positives more correctly. This will increase the recall (true positive rate) since the focus is on identifying all positives. However, this will lead to identifying positives aggressively, and observations 'similar' to 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`.

9.3.1 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,r
    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)

<matplotlib.legend.Legend at 0x25a2695bc10>
```


1.0
0.8
0.6
0.4
0.2
0.0
Precision / Recall
precision
recall
-inf
0
1
2
3
4
5
6
7
8
9
log10(scale_pos_weight)

10 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 - 9. 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 $f_1, ..., f_m$ be the individual models. The expected MSE of an ensemble can be written as:

$$MSE_{Ensemble} = E\left[\left(\frac{1}{m}\sum_{i=1}^{m} f_i(X) - Y\right)^2\right] = \frac{1}{m^2}E\left[\left(f_i(X) - Y\right)^2\right] + \frac{1}{m^2}\sum_{i\neq j} E\left[\left(f_i(X) - Y\right)\left(f_j(X) - Y\right)\right]$$

Assuming the **models are uncorrelated** *(i.e., they have a zero correlation)*, the second term *(covariance of $f_i(.)$ and $f_j(.)$)* reduces to zero, and the expected MSE of the ensemble reduces to:

$$MSE_{Ensemble} = \frac{1}{m}\left(\frac{1}{m}\sum_{i=1}^{m} MSE_{f_i}\right)$$

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, Parame
from sklearn.metrics import mean_squared_error,r2_score,roc_curve,auc,precision_recall_cur
from sklearn.model_selection import KFold
from sklearn.tree import DecisionTreeRegressor,DecisionTreeClassifier
from sklearn.ensemble import VotingRegressor, VotingClassifier, StackingRegressor, Stackin
from sklearn.linear_model import LinearRegression,LogisticRegression, LassoCV, RidgeCV, El
from sklearn.neighbors import KNeighborsRegressor
import itertools as it
import time as time
import xgboost as xgb
from pyearth import Earth
```

```
#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

	carID	brand	model	year	transmission	mileage	fuelType	tax	mpg	engineSize	price
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']
```

10.1 Ensembling regression models

10.1.1 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 XGBoost model from Section 9.2.6
model_xgb = xgb.XGBRegressor(random_state=1,max_depth=8,n_estimators=1000, subsample = 0.7
                                          learning_rate = 0.01,reg_lambda=1, gamma = 100).f
print("RMSE for XGBoost = ", np.sqrt(mean_squared_error(model_xgb.predict(Xtest), ytest)))

#Tuned AdaBoost model from Section 7.2.4
model_ada = AdaBoostRegressor(base_estimator=DecisionTreeRegressor(max_depth=10),n_estimat
                          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), yte

#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),
```

```
RMSE for XGBoost =  5497.553788113875
RMSE for AdaBoost =  5693.165811600585
RMSE for Random forest =  5642.45839697972
RMSE for Gradient Boosting =  5405.787029062213
```

```
#Voting ensemble: Averaging the predictions of all models
en=VotingRegressor(estimators = [('xgb',model_xgb),('ada',model_ada),('rf',model_rf),('gb'
en.fit(X,y)
print("Ensemble model RMSE = ", np.sqrt(mean_squared_error(en.predict(Xtest),ytest)))
```

```
Ensemble model RMSE =  5361.7260763197
```

RMSE of the ensembled model is less than that of each of the individual models.

10.1.2 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.

```
#Stacking using LinearRegression as the metamodel
en = StackingRegressor(estimators = [('xgb', model_xgb),('ada', model_ada),('rf', model_rf
                     final_estimator=LinearRegression(),
                     cv = KFold(n_splits = 5, shuffle = True, random_state=1))
en.fit(X,y)
print("Linear regression metamodel RMSE = ", np.sqrt(mean_squared_error(en.predict(Xtest),
```

```
Linear regression metamodel RMSE =  5311.789386389769
```

```
#Co-efficients of the meta-model
en.final_estimator_.coef_
```

```
array([0.29641759, 0.25626987, 0.051808  , 0.41978153])
```

Note the above coefficients of the meta-model. The model gives the highest weight to the gradient boosting model, and the lowest weight to the random forest model. Also, note that the coefficients need not sum to one.

```
#Stacking using Lasso as the metamodel
en = StackingRegressor(estimators = [('xgb', model_xgb),('ada', model_ada),('rf', model_rf
                     final_estimator=LassoCV(),
                    cv = KFold(n_splits = 5, shuffle = True, random_state=1))
en.fit(X,y)
print("Lasso metamodel RMSE = ", np.sqrt(mean_squared_error(en.predict(Xtest),ytest)))
```

```
Lasso metamodel RMSE =  5311.185592456483
```

```
#Coefficients of the lasso metamodel
en.final_estimator_.coef_
```

```
array([0.17639973, 0.28186944, 0.1152561 , 0.45119952])
```

```
#Stacking using MARS as the meta-model
en = StackingRegressor(estimators = [('xgb',m1),('ada',m2),('rf',m3),('gb',m4)],
                     final_estimator=Earth(max_degree=1),
                    cv = KFold(n_splits = 5, shuffle = True, random_state=1))
en.fit(X,y)
print("MARS metamodel RMSE = ", np.sqrt(mean_squared_error(en.predict(Xtest),ytest)))
```

```
Ensemble model RMSE =  5303.308982301974
```

```
C:\Users\akl0407\Anaconda3\lib\site-packages\pyearth\earth.py:813: FutureWarning: `rcond` pa
To use the future default and silence this warning we advise to pass `rcond=None`, to keep us
  pruning_passer.run()
C:\Users\akl0407\Anaconda3\lib\site-packages\pyearth\earth.py:1066: FutureWarning: `rcond` pa
To use the future default and silence this warning we advise to pass `rcond=None`, to keep us
  coef, resid = np.linalg.lstsq(B, weighted_y[:, i])[0:2]
```

```
print(en.final_estimator_.summary())
```

```
Earth Model
--------------------------------------
Basis Function  Pruned  Coefficient
--------------------------------------
(Intercept)     No      59644
h(x3-75435)     No      0.402779
h(75435-x3)     No      -0.406517
h(x1-74988)     No      0.822699
h(74988-x1)     No      -0.119104
h(x2-72702.8)   No      -0.449716
h(72702.8-x2)   No      -0.280938
x0              No      0.211986
--------------------------------------
MSE: 25038308.7322, GCV: 25226136.6357, RSQ: 0.9070, GRSQ: 0.9063
```

10.2 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 variab
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_
                          DecisionTreeClassifier(max_depth=3),DecisionTreeClassifier(max_d
# 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='ac
# 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'
```

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='ac
# 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'
```

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 neg
             }

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, 'scal
Time taken =  0.9524135629336039  minutes
```

```
#Tuned Adaboost model
model_ada = AdaBoostClassifier(base_estimator=DecisionTreeClassifier(max_depth=1), n_estim
                               random_state=1,learning_rate=0.01).fit(Xtrain, ytrain)
test_accuracy_ada = model_ada.score(Xtest,ytest) #Returns the classification accuracy of t

#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

#Tuned gradient boosting model
model_gb = GradientBoostingClassifier(n_estimators=100, random_state=1,max_depth=4,learnin
                                      subsample = 1.0).fit(Xtrain, ytrain)
```

```
test_accuracy_gb = model_gb.score(Xtest,ytest) #Returns the classification accuracy of the

#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(
test_accuracy_xgb = model_xgb.score(Xtest,ytest) #Returns the classification accuracy of t

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
```

10.2.1 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',mode
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.

10.2.2 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',mode
                                  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.

10.2.3 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',mo
                                    final_estimator=LogisticRegression(random_state=1,max_i
                                    cv = StratifiedKFold(n_splits=5,shuffle=True,random_sta
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 re
ensemble_model = StackingClassifier(estimators=[('ada',model_ada),('rf',model_rf),('gb',mo
                                    final_estimator=RandomForestClassifier(n_estimators=500
                                                                           random_state=1,o
                                    cv = StratifiedKFold(n_splits=5,shuffle=True,random_sta
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_g
                                    final_estimator=RandomForestClassifier(n_estimators=500
                                     random_state=1,oob_score=True),n_jobs=-1,
                                    cv = StratifiedKFold(n_splits=5,shuffle=True,random_sta
    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_fea
oob_score
```

```
{1: 0.8445945945945946,
 2: 0.831081081081081,
 3: 0.8378378378378378,
 4: 0.831081081081081}
```

10.2.4 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',mode

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 = Tru
                          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
```

11 More boosting models

This chapter is not in syllabus. However, the boosting models here may help in the prediction problem and / or in your future data science projects.

```
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_pre
from sklearn.metrics import mean_squared_error,r2_score,roc_curve,auc,precision_recall_cur
recall_score, precision_score, confusion_matrix
from sklearn.tree import DecisionTreeRegressor,DecisionTreeClassifier
from sklearn.model_selection import GridSearchCV, ParameterGrid, StratifiedKFold, Randomiz
from sklearn.ensemble import VotingRegressor, VotingClassifier, StackingRegressor, Stackin
from sklearn.linear_model import LinearRegression,LogisticRegression, LassoCV, RidgeCV, El
from sklearn.neighbors import KNeighborsRegressor
import itertools as it
import time as time
import xgboost as xgb
from pyearth import Earth
from lightgbm import LGBMRegressor
from catboost import CatBoostRegressor
```

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']
```

11.1 LightGBM

LightGBM is a gradient boosting decision tree algorithm developed by Microsoft in 2017. LightGBM outperforms XGBoost in terms of compuational 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.

11.1.1 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.

```
#K-fold cross validation to find optimal parameters for LightGBM regressor
start_time = time.time()
param_grid = {'max_depth': [4,6,8],
              '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(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': 0, 'reg_alpha': 100, 'num_leaves
Optimal cross validation R-squared =  0.8935432951824455
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))
```

```
5400.723918176313
```

11.1.2 LightGBM vs XGBoost

LightGBM model took 1 minute for a random search with 1000 fits as compared to 4 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(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_de
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
```

11.2 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.

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

```
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 RMSE on test data for our example of predicting car prices.

11.2.2 CatBoost vs XGBoost

Let us see the performance of XGBoost with default hyperparameter settings.

```
model_xgb = xgb.XGBRFRegressor().fit(X, y)
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.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.

11.2.3 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],
              'num_leaves': [20, 31, 40],
              'learning_rate': [0.01, 0.05, 0.1],
               'reg_lambda':[0, 10, 100],
                'n_estimators':[100, 500, 1000],
                'subsample': [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=Fa
                                param_distributions = param_grid, n_iter = 200,
                                verbose = 1,random_state = 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

C:\Users\akl0407\Anaconda3\lib\site-packages\sklearn\model_selection\_search.py:918: UserWar
        nan 0.86528582        nan        nan 0.84104458 0.79227627
        nan        nan 0.84395413 0.89887462        nan        nan
        nan 0.90260407        nan        nan        nan        nan
        nan        nan 0.86545114        nan 0.84894322        nan
 0.8913253         nan        nan 0.90681372 0.90270419 0.84033192
        nan        nan        nan        nan        nan 0.89897627
```

```
       nan 0.75750273 0.63799634 0.82429155 0.8541958         nan
 0.85795537 0.84778687        nan 0.82552044 0.88776603        nan
 0.87183014        nan        nan        nan        nan        nan
       nan        nan        nan        nan 0.868381   0.88627774
       nan        nan        nan        nan 0.88302748        nan
       nan        nan 0.87173927 0.90364659        nan 0.68716329
 0.86810108        nan 0.90387934 0.86198    0.79482791        nan
 0.867492          nan        nan        nan 0.8681382         nan
       nan        nan        nan        nan        nan 0.82487077
 0.54242665        nan        nan        nan        nan        nan
       nan        nan 0.88919641        nan        nan        nan
 0.85336326        nan 0.8619873  0.83934649 0.90477081 0.79750609
 0.86543518        nan        nan        nan        nan 0.80115517
       nan        nan        nan        nan        nan 0.75434919
 0.60871141        nan        nan 0.79028956 0.66728925 0.89361737
       nan        nan        nan        nan        nan 0.89080628
       nan 0.75063605        nan 0.90090587        nan        nan
 0.82573579 0.90680318 0.85290443        nan        nan 0.89928321
       nan        nan        nan 0.86285405 0.8978184         nan
       nan 0.84783232        nan        nan        nan        nan
 0.86225177        nan        nan 0.8621329         nan        nan
 0.54359637        nan        nan 0.8994749  0.84800071        nan
       nan        nan        nan        nan        nan        nan
 0.63020005        nan 0.87308398        nan 0.86614844        nan
       nan        nan        nan        nan        nan        nan
 0.68846678 0.8406747         nan        nan        nan        nan
 0.88741464 0.86148835]
  warnings.warn(

Optimal parameter values = {'subsample': 0.75, 'reg_lambda': 0, 'num_leaves': 31, 'n_estimato
Optimal cross validation R-squared =  0.9068137174802073
Time taken =  2  minutes
```

```
#RMSE based on the optimal parameter values
np.sqrt(mean_squared_error(optimal_params.best_estimator_.predict(Xtest),ytest))
```

```
5254.902079026533
```

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 LighGBM, and likely to require lesser tuning as compared to XGBoost.

A Assignment A

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. Do not write your name on the assignment.
3. Write your code in the *Code* cells and your answer in the *Markdown* cells of the Jupyter notebook. Ensure that the solution is written neatly enough to understand and grade.
4. Use Quarto to print 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.
5. The assignment is worth 100 points, and is due on **Thursday, 13th April 2023 at 11:59 pm**.
6. **Four points are properly formatting the assignment**. The breakdown is as follows:

- Must be an HTML file rendered using Quarto (1 pt). *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. If your issue doesn't seem genuine, you will lose points.*
- 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 pt)
- Final answers of each question are written in Markdown cells (1 pt).
- There is no piece of unnecessary / redundant code, and no unnecessary / redundant text (1 pt)

A.1 Bias-variance trade-off

Throughout the course, the conceptual clarity about bias and variance will help you tune the models for optimal performance and enable you to compare different models in terms of

bias and variance. In this question, you will perform simulations to understand and visualize bias-variance trade-off as in Fig. 2.12 of the book (page 36).

Assume that the response y is a function of the predictors x_1 and x_2 and includes a random error ϵ, as follows:

$$y = f(x_1, x_2) + \epsilon, \tag{A.1}$$

where the function $f(.)$ is the Bukin function, $x_1 \sim U[-15, -5], x_2 \sim U[-3, 3]$, and $\epsilon \sim N(0, \sigma^2); \sigma = 10$. *Here U refers to Uniform distribution, and N refers to normal distribution.* Use NumPy to simulate values from these distributions.

You will code an algorithm (described below) to compute the expected squared bias, expected variance, var(ϵ) and expected test MSE of the following 7 linear regression models having the predictors as:

1. x_1 and x_2
2. All the predictors in the above model, and all polynomial combinations of x_1, and x_2 of degree 2, which will be x_1^2, x_2^2, and $x_1 x_2$
3. All the predictors in the above model, and all polynomial combinations of x_1, and x_2 of degree 3, which will be $x_1^3, x_2^3, x_1^2 x_2$, and $x_1 x_2^2$
4. All the predictors in the above model, and all polynomial combinations of x_1, and x_2 of degree 4
5. All the predictors in the above model, and all polynomial combinations of x_1, and x_2 of degree 5
6. All the predictors in the above model, and all polynomial combinations of x_1, and x_2 of degree 6
7. All the predictors in the above model, and all polynomial combinations of x_1, and x_2 of degree 7

As you can see the models are arranged in increasing order of flexibility / complexity. This corresponds to the horizontal axis of Fig. 2.12 in the book.

Use the following **algorithm** to compute the expected squared bias, expected variance, var(ϵ) and expected test MSE of the 7 linear regression models above:

I. Define the Bukin function that accepts x_1 and x_2 as parameters and returns the Bukin function value ($f(x_1, x_2)$).

(2 points)

II. Repeat steps **III - VII** for all degrees d in $\{1, 2, ..., 7\}$

(2 points)

III. Considering a model of **degree** d, simulate the following test and train datasets.

A. **Simulate test data**

1. Set a seed of 100. Use the code: `np.random.seed(100)`, where `np` refers to the numpy library
2. Simulate 100 values of x_1 from $U[-15, -5]$.
3. Simulate 100 values of x_2 from $U[-3, 3]$.
4. Compute the Bukin function value $f(x_1, x_2)$ for the simulated values of x_1 and x_2.
5. Use the function `PolynomialFeatures` from the `preprocessing` module of the `sklearn` library to create all polynomial combinations of x_1, and x_2 up to degree d.

(4 points)

B. **Simulate 100 train data sets**, where each train data is simulated as follows:

1. Set a seed of i for simualting the *ith* train data. Use the code: `np.random.seed(i)`, where `np` refers to the numpy library.
2. Simulate 100 values of x_1 from $U[-15, -5]$
3. Simulate 100 values of x_2 from $U[-3, 3]$
4. Compute the Bukin function value $f(x_1, x_2)$ for the simulated values of x_1 and x_2
5. Simulate the response y using the above set of simulated values with Equation A.1
6. Use the function `PolynomialFeatures` from the `preprocessing` module of the `sklearn` library to create all polynomial combinations of x_1, and x_2 up to degree d.

(6 points)

IV. For each train data in III(B), develop a linear regression model using the `LinearRegression()` function from the `linear_model` module of the `sklearn` library.

(2 points)

V. Note that the squared bias at a test point x_{1_test}, x_{2_test} is:

$$[Bias(\hat{f}(x_{1\_test}, x_{2\_test}))]^2 = [E(\hat{f}(x_{1\_test}, x_{2\_test})) - f(x_{1\_test}, x_{2\_test})]^2, \tag{A.2}$$

where $E(\hat{f}(x_{1\_test}, x_{2\_test}))$ is the mean prediction of the 100 trained models at x_{1_test}, x_{2_test}.

Compute the overall expected squared bias as the average squared bias at all the test data points, as in the equation below:

$$[Bias(\hat{f}(.))]^2 = \frac{1}{100}\Sigma_{i=1}^{100}[Bias(\hat{f}(x_{1i\_test}, x_{2i\_test}))]^2, \tag{A.3}$$

(8 points)

VI. Note that the variance at a test point x_{1_test}, x_{2_test} is $Var(\hat{f}(x_{1\_test}, x_{2\_test}))$. Compute the overall expected variance as the average variance at all the test data points, as in the equation below:

$$Var(\hat{f}(.)) = \frac{1}{100}\Sigma_{i=1}^{100}Var(\hat{f}(x_{1i\_test}, x_{2i\_test})) \tag{A.4}$$

(6 points)

VII. Compute the overall expected test mean squared error as the sum of the expected squared bias (Equation A.3), expected variance (Equation A.4), and error variance (σ^2):

$$MSE = [Bias(\hat{f}(.))]^2 + Var(\hat{f}(.)) + \sigma^2, \tag{A.5}$$

(4 points)

VIII. Plot the overall expected squared bias, overall expected variance, and overall expected test MSE *(as obtained from Equation A.3, Equation A.4, and Equation A.5 respectively)* against the degree d (or flexibility / complexity) of the model . Your plot should look like one of the plots in Fig. 2.12 of the book.

(3 points)

IX. What is the degree of the optimal model, i.e., the degree that provides the best **bias-variance trade-off**?

(2 points)

Note: While coding the algorithm, comment it well so that it is easy to give partial credit in case of mistakes. Include the numerals of the algorithm (such as II(B), V, VI, etc.) in your comments so that it is easy to check your algorithm for completeness.

A.2 Tuning a classification model with `sklearn`

Data

Read the data *classification_data.csv*. The description of the columns is as follows:

1. `hi_int_prncp_pd`: Indicates if a high percentage of the repayments made went to interest rather than principal. **Target variable.**
2. `out_prncp_inv`: Remaining outstanding principal for portion of total amount funded by investors
3. `loan_amnt`: The listed amount of the loan applied for by the borrower. If at some point in time, the credit department reduces the loan amount, then it will be reflected in this value.
4. `int_rate`: Interest rate on the loan
5. `term`: The number of payments on the loan. Values are in months and can be either 36 or 60.

You will develop and tune a logistic regression model to predict `hi_int_prncp_pd` based on the rest of the columns (predictors) as per the instructions below.

A.2.1 Train-test split

Use the function `train_test_split` from the `model_selection` module of the `sklearn` library to split the data into 75% train and 25% test. Stratify the split based on the response. Use `random_state` as 45. Print the proportion of 0s and 1s in both the train and test datasets.

(4 points)

A.2.2 Scaling predictors

Scale the predictors to avoid convergence errors when fitting the logistic regression model.

Note that last quarter, we were focusing on inference (along with prediction), so we avoided scaling. It is a bit inconvenient to interpret odds with scaled predictors. However, avoiding scaling may lead to convergence errors as some of you saw in your course projects. So, it is a good practice to scale, especially when your focus is prediction.

(3 points)

A.2.3 Tuning the degree

Use the functions:

1. `cross_val_score` from the `model_selection` module of the `sklearn` library to tune the degree of the logistic regression model for maximizing the stratified 5-fold prediction accuracy. Consider degrees from 1 to 6.
2. `PolynomialFeatures` from the `preprocessing` module of the `sklearn` library to create all polynomial combinations of the predictors up to degree d.

What is the optimal degree?

(4 points)

Notes:

- A model of degree d will consist of polynomial transformations and interactions of predictors up to degree d. For example, a model of degree 2 will consist of the square of each predictor and all 2-factor interactions of the predictors.
- You may use the `newton-cg` solver to avoid convergence issues.
- Use the default `C` value at this point, you will tune it later.

A.2.4 Test accuracy with optimal degree

For the optimal degree identified in the previous question, compute the test accuracy.

(4 points)

A.2.5 Tuning `C`

With the optimal degree identified in the previous question, find the optimal regularization parameter `C`. Again use the `cross_val_score` function.

(3 points)

A.2.6 Test accuracy with optimal degree and `C`

For the optimal degree and optimal `C` identified in the previous questions, compute the test accuracy.

(3 points)

A.2.7 Tuning decision threshold probability

With the optimal degree and optimal `C` identified in the previous questions, find the optimal decision threshold probability to maximize accuracy. Use the `cross_val_predict` function.

(4 points)

A.2.8 Test accuracy for optimal degree, `C`, and threshold probability

For the optimal degree, optimal `C`, and optimal decision threshold probabilities identified in the previous questions, compute the test accuracy.

(4 points)

A.2.9 Simultaneous optimization of multiple parameters

In the above tuning approach we optimized the hyperparameters and the decision threshold probability sequentially. This is a greedy approach, which doesn't consider all combinations of hyperparameters and decision threshold probabilities, and thus may fail to find the optimal combination of values that maximize accuracy. Thus, tune both the model hyperparameters - degree and `C`, and the decision threshold probability simultaneously considering all value combinations. This will take more time, but is likely to provide more accurate optimal parameter values.

(6 points)

A.2.10 Test accuracy with optimal parameters obtained simultaneously

For the optimal degree, optimal `C`, and optimal decision threshold probabilities identified in the previous question, compute the test accuracy.

(4 points)

A.2.11 Optimizing parameters for multiple performance metrics

Find the optimal `C` and degree to maximize recall while having a precision of more than 75%. Use the function `cross_validate` from the `model_selection` module of the `sklearn` library.

Note: `cross_validate` function is very similar to `cross_val_score`, the only difference is you can use multiple metrics with the scoring input, as you need in this question.

(8 points)

A.2.12 Performance metrics computation

For the optimal degree and `C` identified in the previous question, compute the following performance metrics on test data. Use `sklearn` functions, manual computation is not allowed.

1. Precision
2. Recall
3. Accuracy
4. ROC-AUC
5. Show the confusion matrix

(10 points)

B Assignment B

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. Do not write your name on the assignment.
3. Write your code in the *Code* cells and your answer in the *Markdown* cells of the Jupyter notebook. Ensure that the solution is written neatly enough to understand and grade.
4. Use Quarto to print 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.
5. The assignment is worth 100 points, and is due on **Sunday, 23rd April 2023 at 11:59 pm**.
6. **Five points are properly formatting the assignment**. The breakdown is as follows:

- Must be an HTML file rendered using Quarto (2 pts). *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. If your issue doesn't seem genuine, you will lose points.*
- 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 pt)
- Final answers of each question are written in Markdown cells (1 pt).
- There is no piece of unnecessary / redundant code, and no unnecessary / redundant text (1 pt)

7. For all questions on cross-validation, you must use `sklearn` functions.

B.1 Degrees of freedom

Find the number of degrees of freedom of the following models. Exclude the intercept when counting the degrees of freedom. You may either show your calculation, or explain briefly how you are computing the degrees of freedom.

B.1.1 Quadratic spline

A model with one predictor, where the predictor is transformed into a quadratic spline with 5 knots

(2 points)

B.1.2 Natural cubic splines

A model with one predictor, where the predictor is transformed into a natural cubic spline with 4 knots

(2 points)

B.1.3 Generalized additive model

A model with four predictors, where the transformations of the respective predictors are (i) cubic spline transformation with 3 knots, (ii) log transformation, (iii) linear spline transformation with 2 knots, (iv) polynomial transformation of degree 4.

(4 points)

B.2 Number of knots

Find the number of knots in the following spline transformations, if each of the transformations corresponds to 7 degrees of freedom (excluding the intercept).

B.2.1 Cubic splines

Cubic spline transformation

(1 point)

B.2.2 Natural cubic splines

Natural cubic spline transformation

(1 point)

B.2.3 Degree 4 spline

Spline transformation of degree 4

(1 point)

B.3 Regression problem

Read the file *investment_clean_data.csv*. This data is a cleaned version of the file *train.csv* in last quarter's regression prediction problem. Refer to the link for description of variables. It required some effort to get a RMSE of less than 650 with linear regression. In this question, we'll use MARS / natural cubic splines to get a RMSE of less than 350 with relatively less effort. Use mean squared error as the performance metric in cross validation.

B.3.1 Data preparation

Prepare the data for modeling as follows:

1. Use the Pandas function `get_dummies()` to convert all the categorical predictors to dummy variables.
2. Using the `sklearn` function `train_test_split`, split the data into 20% test and 80% train. Use `random_state = 45`.

Note:

A. The function `get_dummies()` can be used over the entire DataFrame. Don't convert the categorical variables individually.

B. The MARS model does not accept categorical predictors, which is why the conversion is done.

C. The response is `money_made_inv`

(2 points)

B.3.2 Optimal MARS degree

Use 5-fold cross validation to find the optimal degree of the MARS model to predict `money_made_inv` based on all the predictors in the dataset.

Hint: Start from degree 1, and keep going until it doesn't benefit.

(4 points)

B.3.3 Fitting MARS model

With the optimal degree identified in the previous question, fit a MARS model. Print the model summary. What is the degree of freedom of the model (excluding the intercept)?

(1 + 1 + 2 points)

B.3.4 Interpreting MARS basis functions

Based on the model summary in the previous question, answer the following question. Holding all other predictors constant, what will be the mean increase in `money_made_inv` for a unit increase in `out_prncp_inv`, given that `out_prncp_inv` is in [500, 600], `term` = 36 (months), `loan_amnt` = 1000, and `int_rate` = 0.1?

First, write the basis functions being used to answer the question, and then substitute the values.

Also, which basis function is non-zero for the smallest domain space of `out_prncp_inv`? Also, specify the domain space in which it is non-zero.

(3 + 2 points)

B.3.5 Feature importance

Find the relative importance of each predictor in the MARS model developed in B.3.3. You may choose any criterion for finding feature importance based on the MARS documentation. Print a DataFrame with 2 columns - one column consisting of predictors arranged in descending order of relative importance, and the second column quantifying their relative importance. Exclude predictors rejected by the model developed in B.3.3.

Note the forward pass and backward passes of the algorithm perform feature selection without manual intervention.

(4 points)

B.3.6 Prediction

Using the model developed in B.3.3, compute the RMSE on test data.

(2 points)

Non-trivial train data

Let us call the part of the dataset where `out_prncp_inv = 0` as a trivial subset of data. For this subset, we can directly predict the response without developing a model *(recall the EDA last quarter)*. For all the questions below, fit / tune the model only on the non-trivial part of the train data. However, when making predictions, and computing RMSE, consider the entire test data. Combine the predictions of the model on the non-trivial subset of test data with the predictions on the trivial subset of test data to make predictions on the entire test data.

B.3.7 Prediction with non-trivial train data

Find the optimal degree of the MARS model based on the non-trivial train data, fit the model, and re-compute the RMSE on test data.

Note: You should get a lesser RMSE as compared to what you got in B.3.6.

(4 points)

B.3.8 Reducing model variance

The MARS model is highly flexible, which makes it a low bias-high variance model. However, high prediction variance increases the expected mean squared error on test data *(see **equation 2.7 on page 34** of the book)*. How can you reduce the prediction variance of the model without increasing the bias? Check slide 12 of the bias-variance presentation. The MARS model, in general, corresponds to case B. You can see that by averaging the predictions of multiple models, you will reduce prediction variance without increasing the bias.

Take 10 samples of train data of the same size as the train data, with replacement. For each sample, fit a MARS model with the optimal degree identified earlier. Use the i^{th} model, say $\hat{f}_i$ to make prediction $\hat{f}_i(\mathbf{x}_{test})$ on each test data point $\mathbf{x}_{test}$ *(Note that predictions will be made using the model on the non-trivial test data, and without the model on the trivial test data)*. Compute the average prediction on each test data point based on the 10 models as follows:

$$\hat{f}(\mathbf{x}_{test}) = \frac{1}{10}\Sigma_{1=1}^{10}\hat{f}_i(\mathbf{x}_{test})$$

Consider $\hat{f}(\mathbf{x}_{test})$ as the prediction at the test data point $\mathbf{x}_{test}$. Compute the RMSE based on this model, which is the average prediction of 10 models. You should get a lesser RMSE as compared to the previous question (B.3.7).

Note: For ease in grading, use the Pandas DataFrame method `sample` to take samples with replacement, and put `random_state` for the ith sample as i, where i goes from 0 to 9.

(6 points)

B.3.9 Generalized additive model (GAM)

Develop a Generalized linear model $\hat{f}_{GLM}(.)$ to predict `money_made_inv` as follows:

$$\hat{f}_{GLM}(\mathbf{x}) = \hat{\beta}_0 + \Sigma_{i=1}^{4} \hat{\beta}_i f_i(\mathbf{x}),$$

where $f_i(\mathbf{x})$ is a MARS model of degree i.

Print the estimated beta coefficients $(\hat{\beta}_0, \hat{\beta}_1, \hat{\beta}_2, \hat{\beta}_3, \hat{\beta}_4)$ of the developed model.

Note: The model is developed on the non-trivial train data

(8 points)

B.3.10 Prediction with GAM

Use the GAM developed in the previous question to compute RMSE on test data.

Note: Predictions will be made using the model on the non-trivial test data, and without the model on the trivial test data

(5 points)

B.3.11 Reducing GAM prediction variance

As we reduced the variance of the MARS model in B.3.8, follow the same approach to reduce the variance of the GAM developed in B.3.9, and compute the RMSE on test data.

Note: You should get a lesser RMSE as compared to what you got in B.3.10.

(8 points)

B.3.12 Natural cubic splines

Even though MARS is efficient and highly flexible, natural cubic splines work very well too, if tuned properly.

Consider the predictors identified in the model summary of the MARS model printed in B.3.3. For each predictor, create natural cubic splines basis functions with d degrees of freedom. Include all-order interactions *(i.e., 2-factor, 3-factor, 4-factor interactions, and so on)* of all the basis functions. Use the `sklearn` function `cross_val_score()` to find and report the optimal degrees of freedom for the natural cubic spline of each predictor.

Consider degrees of freedom from 3 to 6 for the natural cubic spline transformation of each predictor.

(8 points)

B.3.13 Fitting the natural cubic splines model

With the optimal degrees of freedom identified in the previous question, fit a model to predict `money_made_inv`, where the basis functions correspond to the natural cubic splines of each predictor, and all-factor interactions of the basis functions. Compute the RMSE on test data.

Note: Predictions will be made using the model on the non-trivial test data, and without the model on the trivial test data

(4 points)

B.4 GAM for classification

The data for this question is related with direct marketing campaigns of a Portuguese banking institution. The marketing campaigns were based on phone calls, where bank clients were called to subscribe for a term deposit.

There is one train data - *train.csv*, which you will use to develop a model. There are two test datasets - *test1.csv* and *test2.csv*, which you will use to test your model. Each dataset has the following attributes about the clients called in the marketing campaign:

1. `age`: Age of the client
2. `education`: Education level of the client
3. `day`: Day of the month the call is made
4. `month`: Month of the call
5. `y`: did the client subscribe to a term deposit?

6. `duration`: Call duration, in seconds. This attribute highly affects the output target (e.g., if `duration`=0 then `y`='no'). Yet, the duration is not known before a call is performed. Also, after the end of the call `y` is obviously known. Thus, this input should only be included for inference purposes and should be discarded if the intention is to have a realistic predictive model.

(Raw data source: Source. Do not use the raw data source for this assignment. It is just for reference.)

Develop a **generalized additive model (GAM)** to predict the probability of a client subscribing to a term deposit based on *age, education, day* and *month*. The model must have:
(a) **Minimum overall classification accuracy of 75%** among the classification accuracies on *train.csv*, *test1.csv* and *test2.csv*.
(b) **Minimum recall of 55%** among the recall on *train.csv*, *test1.csv* and *test2.csv*.

Print the accuracy and recall for all the three datasets - *train.csv*, *test1.csv* and *test2.csv*.

Note that:

i. You cannot use `duration` as a predictor. The predictor is not useful for prediction because its value is determined after the marketing call ends. However, after the call ends, we already know whether the client responded positively or negatively.

ii. One way to develop the model satisfying constrains (a) and (b) is to use **spline transformations for *age* and *day*, and interacting *month* with all the predictors (including the spline transformations)**

iii. You may assume that the distribution of the predictors is the same in all the three datasets. Thus, you may create B-spline basis functions independently for the train and test datasets.

iv. Use cross-validation on train data to optimize the model hyperparameters, and the decision threshold probability. Then, use the optimal hyperparameters to fit the model on train data. Then, evaluate its accuracy and recall on all the three datasets. Note that the test datasets must only be used to evaluate performance metrics, and not optimize any hyperparameters or decision threshold probability.

(20 points: 10 points for cross validation, 5 points for obtaining and showing the optimal values of the hyperparameters and decision threshold probability, 2 points for fitting the model with the optimal hyperparameters, and 3 points for printing the accuracy & recall on each of the three datasets)

C Assignment C

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 answer in the *Markdown* cells of the Jupyter notebook. Ensure that the solution is written neatly enough to understand and grade.
3. Use Quarto to print 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 **Sunday, 7th May 2023 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 pts). *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. If your issue doesn't seem genuine, you will lose points.*
- 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 pt)
- Final answers of each question are written in Markdown cells (1 pt).
- There is no piece of unnecessary / redundant code, and no unnecessary / redundant text (1 pt)

C.1 Regression Problem - Miami housing

C.1.1 Data preparation

Read the data *miami-housing.csv*. Check the description of the variables here. Split the data into 60% train and 40% test. Use `random_state = 45`. The response is `SALE_PRC`, and the rest

of the columns are predictors, except `PARCELNO`. Print the shape of the predictors dataframe of the train data.

(2 points)

C.1.2 Decision tree

Develop a decision tree model to predict `SALE_PRC` based on all the predictors. Use `random_state = 45`. Use the default hyperparameter values. What is the MAE (mean absolute error) on test data?

(3 points)

C.1.3 Tuning decision tree

Tune the hyperparameters of the decision tree model developed in the previous question, and compute the MAE on test data. You must tune the hyperparameters in the following manner:

1. Use `GridSearchCV` to minimize the 5-fold mean absolute error (MAE).
2. Use must do a coarse grid search first to get an idea of the domain space where the optimal hyperparameter values lie.
3. You must follow it up with a finer grid search to get more precise optimal hyperparameter values.
4. You may decide yourself which hyperparameters you wish to tune. Common sense should help. There is no single correct answer.

The MAE must be less than $66,000. You must show the optimal values of the hyperparameters obtained, and the test MAE.

(3 points for coarse grid search, 3 points for finer grid search, 4 points for reaching the required MAE)

C.1.4 Bagging decision trees

Bag decision trees, and compute the MAE on test data. Use enough number of trees, such that the MAE stabilizes. Other than `n_estimators`, use default values of hyperparameters.

The test MAE must be less than $50,000.

(4 points)

C.1.5 Bagging without bootstrapping

Bag decision trees without bootstrapping, i.e., put `bootstrap = False` while bagging the trees, and compute the MAE on test data. Why is the MAE obtained much higher than that in the previous question, but lower than that obtained in C.1.2?

(1 point for code, (3 + 3) points for reasoning)

C.1.6 Tuning bagged tree model

C.1.6.1 Approaches

There are two approaches for tuning a bagged tree model:

1. Out of bag prediction
2. K-fold cross validation using `GridSearchCV`.

What is the advantage of each approach over the other, i.e., what is the advantage of the out-of-bag approach over K-fold cross validation, and what is the advantage of K-fold cross validation over the out-of-bag approach?

(3 + 3 points)

C.1.6.2 Tuning the hyperparameters

Tune the hyperparameters of the bagged tree model developed in C.1.4. You may use either of the approaches mentioned in the previous question. Show the optimal values of the hyperparameters obtained. Compute the MAE on test data with the tuned model. Your MAE on test data must be less than \$46,000. However, you **cannot use the test data to tune the hyperparameters**.

It is up to you to pick the hyperparameters and their values in the grid.

(10 points)

C.1.7 Bagging feature importance

Arrange and print the predictors in decreasing order of importance.

(4 points)

C.1.8 Random forest

C.1.8.1 Tuning random forest

Tune a random forest model to predict `SALE_PRC`, and compute the MAE on test data. The MAE must be less than $46,000.

It is up to you to pick the hyperparameters and their values in the grid.

(10 points)

C.1.8.2 Feature importance

Arrange and print the predictors in decreasing order of importance.

(4 points)

C.1.8.3 Random forest vs bagging: `max_features`

Note that the `max_features` hyperparameter is there both in the `RandomForestRegressor()` function and the `BaggingRegressor()` function. Does it have the same meaning in both the functions? If not, then what is the difference?

Hint: Check scikit-learn documentation

(1 + 3 points)

C.2 Classification - Term deposit

The data for this question is related with direct marketing campaigns of a Portuguese banking institution. The marketing campaigns were based on phone calls, where bank clients were called to subscribe for a term deposit.

There is a train data - *train.csv*, which you will use to develop a model. There is a test data - *test.csv*, which you will use to test your model. Each dataset has the following attributes about the clients called in the marketing campaign:

1. `age`: Age of the client
2. `education`: Education level of the client
3. `day`: Day of the month the call is made
4. `month`: Month of the call

5. `y`: did the client subscribe to a term deposit?
6. `duration`: Call duration, in seconds. This attribute highly affects the output target (e.g., if `duration`=0 then `y`='no'). Yet, the duration is not known before a call is performed. Also, after the end of the call `y` is obviously known. Thus, this input should only be included for inference purposes and should be discarded if the intention is to have a realistic predictive model.

(Raw data source: Source. Do not use the raw data source for this assignment. It is just for reference.)

C.2.1 Data preparation

Convert all the categorical predictors in the data to dummy variables. Note that `month` and `education` are categorical variables.

(2 points)

C.2.2 Decision tree

Develop and tune a **decision tree model** to predict the probability of a client subscribing to a term deposit based on `age`, `education`, `day` and `month`. The model must have:

(a) **Minimum overall classification accuracy of 70%** among the classification accuracies on *train.csv*, and *test.csv*.

(b) **Minimum recall of 60%** among the recall on *train.csv*, and *test.csv*.

Print the accuracy and recall for both the datasets - *train.csv*, and *test.csv*.

Note that:

i. You cannot use `duration` as a predictor. The predictor is not useful for prediction because its value is determined after the marketing call ends. However, after the call ends, we already know whether the client responded positively or negatively.

ii. You are free to choose any value of threshold probability for classifying observations. However, you must use the same threshold on both the datasets.

iii. Use cross-validation on train data to optimize the model hyperparameters.

iv. Using the optimal model hyperparameters obtained in (iii), develop the decision tree model. Plot the 5-fold cross-validated accuracy and recall against decision threshold probability. Tune the decision threshold probability based on the plot, or the data underlying the plot to achieve the required trade-off between recall and accuracy.

v. Evaluate the accuracy and recall of the developed model with the tuned decision threshold probability on both the datasets. Note that the test dataset must only be used to evaluate performance metrics, and not optimize any hyperparameters or decision threshold probability.

(14 points - 4 points for tuning the hyperparameters, 4 points for making the plot, 4 points for tuning the decision threshold probability based on the plot, and 2 points for printing the accuracy & recall on both the datasets)

Hint: Restrict the search for `max_depth` to a maximum of 25, and `max_leaf_nodes` to a maximum of 50. Without this restriction, you may get a better recall for threshold probability = 0.5, but are likely to get a worse trade-off between recall and accuracy.

It is up to you to pick the hyperparameters and their values in the grid.

C.2.3 Random forest

Develop and tune a **random forest model** to predict the probability of a client subscribing to a term deposit based on `age`, `education`, `day` and `month`. The model must have:

(a) **Minimum overall classification accuracy of 75%** among the classification accuracies on *train.csv*, and *test.csv*.

(b) **Minimum recall of 60%** among the recall on *train.csv*, and *test.csv*.

Print the accuracy and recall for both the datasets - *train.csv*, and *test.csv*.

Note that:

i. You cannot use `duration` as a predictor. The predictor is not useful for prediction because its value is determined after the marketing call ends. However, after the call ends, we already know whether the client responded positively or negatively.

ii. You are free to choose any value of threshold probability for classifying observations. However, you must use the same threshold on both the datasets.

iii. Use cross-validation on train data to optimize the model hyperparameters.

iv. Using the optimal model hyperparameters obtained in (iii), develop the decision tree model. Plot the cross-validated accuracy and recall against decision threshold probability. Tune the decision threshold probability based on the plot, or the data underlying the plot to achieve the required trade-off between recall and accuracy.

v. Evaluate the accuracy and recall of the developed model with the tuned decision threshold probability on both the datasets. Note that the test dataset must only be used to evaluate performance metrics, and not optimize any hyperparameters or decision threshold probability.

(12 points - 4 points for tuning the hyperparameters, 3 points for making the plot, 3 points for tuning the decision threshold probability based on the plot, and 2 points for printing the accuracy & recall on both the datasets)

Hint: Restrict the search for `max_depth` to a maximum of 25, and `max_leaf_nodes` to a maximum of 45. Without this restriction, you may get a better recall for threshold probability = 0.5, but are likely to get a worse trade-off between recall and accuracy.

It is up to you to pick the hyperparameters and their values in the grid.

C.3 Predictor transformations in trees

Can a non-linear monotonic transformation of predictors (such as *log(), sqrt()* etc.) be useful in improving the accuracy of decision tree models?

(3 points for answer)

D Assignment D

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 answer in the *Markdown* cells of the Jupyter notebook. Ensure that the solution is written neatly enough to understand and grade.
3. Use Quarto to print 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 **Friday, 19th May 2023 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 pts).
- 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 pt)
- Final answers of each question are written in Markdown cells (1 pt).
- There is no piece of unnecessary / redundant code, and no unnecessary / redundant text (1 pt)

D.1 Conceptual

D.1.1 AdaBoost vs Random Forest

Among AdaBoost and Random Forest, which model is more sensitive to outliers in response and why? Consider both regression and classification.

(1 + 3 points)

D.1.2 Loss functions

Which loss functions should you use in boosting algorithms to reduce sensitivity to outliers in response, as compared to the squared error loss function, for regression problems. Name any 2 loss functions, and explain how they reduce the sensitivity towards outliers.

(2 + 2 points)

D.2 Regression Problem - Miami housing

D.2.1 Data preparation

Read the data *miami-housing.csv*. Check the description of the variables here. Split the data into 60% train and 40% test. Use `random_state = 45`. The response is `SALE_PRC`, and the rest of the columns are predictors, except `PARCELNO`. Print the shape of the predictors dataframe of the train data.

(2 points)

D.2.2 AdaBoost hyperparameter tuning

Develop and tune an AdaBoost model to predict `SALE_PRC` based on all the predictors. Compute the MAE on test data.

You must tune in the following manner:

1. Use `GridSearchCV` to minimize the 5-fold mean absolute error (MAE).
2. You are advised to do a coarse grid search first to get an idea of the domain space where the optimal hyperparameter values lie.
3. If you reach the goal with the coarse grid search, you can stop. Otherwise, you may follow it up with a finer grid search to get more precise optimal hyperparameter values.
4. You may decide yourself which hyperparameters you wish to tune. Tuning `max_depth`, `n_estimators`, and `learning_rate` should suffice.

The test MAE must be less than **$46,000**. You must show the optimal values of the hyperparameters obtained, and the test MAE.

Note: Hyperparameter tuning must be done on train data. Test data is only to assess model performance. Test data must remain untouched until the model is finalized, and must only be used to compute the test MAE.

Hint: Below is one way to solve the problem. Note that there may be several completely different and better ways to solve the problem.

1. Consider tree depths of 3, 5, and 10, number of trees as 10, 50, 100, and 200, and learning rates as 0.0001, 0.001, 0.01, 0.1, and 1.0. `GridSearchCV` takes 2 minutes to execute on a 6-core laptop for these values.
2. With the above search, you will probably fail to achieve the objective. However, when you visualize the 5-fold MAE with each of the hyperparameter values considered, you will realize that there is a particular hyperparameter for which you should consider higher / lower values. You will also realize that you need not consider some of the values of the remaining hyperparameters.
3. Do another 2-minute grid search based on what you realized in (2), and you should achieve the objective.

(10 points)

D.2.3 AdaBoost feature importance

Arrange and print the predictors in decreasing order of importance.

(2 points)

D.2.4 Huber loss

What is the advantage of the Huber loss function *(page 349 of Elements of Statistical Learning)* over the squared error and absolute error loss functions?

(4 points)

D.2.5 `RandomizedSearchCV` vs `GridSearchCV`

What's the advantage of `GridSearchCV` over `RandomizedSearchCV` and vice-versa? When will `GridSearchCV` be preferred over `RandomizedSearchCV` and vice-versa?

(4 points)

D.2.6 Gradient boosting (Huber loss) hyperparameter tuning

Develop and tune a Gradient boosting model with `Huber` loss to predict `SALE_PRC` based on all the predictors. Compute the MAE on test data.

You must tune in the following manner:

1. Use may use `GridSearchCV` or `RandomizedSearchCV` to minimize the K-fold mean absolute error (MAE). You may choose any K.
2. You may decide yourself which hyperparameters you wish to tune. Tuning `max_depth`, `n_estimators`, `learning_rate`, and `subsample` should suffice.

The MAE must be less than **$43,000**. You must show the optimal values of the hyperparameters obtained, and the test MAE.

Note: Hyperparameter tuning must be done on train data. Test data is only to assess model performance. Test data must remain untouched until the model is finalized, and must only be used to compute the test MAE.

Hint: Below is one way to solve the problem. Note that there may be several completely different and better ways to solve the problem.

1. Use 2-fold cross-validation to make the execution speed higher. Here, we are compromising - adding bias to the CV error to get a lesser execution time.
2. In gradient boosting, the suggested depth of trees is in [4, 8] *(see page 363 in Elements of Statistical Learning)*. So, consider depths of 4, 6, and 8. Consider 3 values of number of trees in [100, 1000], 3 values of learning rates in [0.01, 0.5], and 3 subsample values in [0.5, 1]. It takes 8 minutes on a 6-core laptop to do an exhaustive search on these values.
3. With the above search, you will probably fail to achieve the objective. However, when you compare the optimal hyperparameter values obtained with the hyperparameter values considered, you will realize that there are some hyperparameters for which you should consider higher / lower values.
4. Do another 10-minute grid search based on what you realized in (3), and you should achieve the objective.
5. Further fine-tuning may reduce your MAE to up to $42,400. However, you can stop once it is below \$43,000 in (4).

(14 points)

D.2.7 Gradient boosting feature importance

Arrange and print the predictors in decreasing order of importance.

(2 points)

D.2.8 Bias-variance

For each of the following hyperparameters tuned in the previous question, explain how do they effect the bias / variance of a gradient boosting model, when their values are increased.

D.2.8.1 `max_depth`

D.2.8.2 `n_estimators`

D.2.8.3 `learning_rate`

D.2.8.4 `subsample`

(8 points)

D.2.9 XGBoost objective function

How is XGboost different from gradient boosting performed with the `GradientBoostingRegressor()` function in the previous question with regard to the optimization objective? How does it benefit prediction accuracy with XGBoost?

(4 points)

D.2.10 XGBoost hyperparameter tuning

Develop and tune an XGBoost model to predict `SALE_PRC` based on all the predictors. Compute the MAE on test data.

You must tune in the following manner:

1. Use may use `GridSearchCV` or `RandomizedSearchCV` to minimize the K-fold mean absolute error (MAE). You may choose any K.
2. You may decide yourself which hyperparameters you wish to tune. Tuning `max_depth`, `n_estimators`, `learning_rate`, `reg_lambda`, `gamma` and `subsample` should suffice.

The test MAE must be less than **$42,000**. You must show the optimal values of the hyperparameters obtained, and the test MAE.

Note: Hyperparameter tuning must be done on train data. Test data is only to assess model performance. Test data must remain untouched until the model is finalized, and must only be used to compute the test MAE.

Hint: Below is one way to solve the problem. Note that there may be several completely different and better ways to solve the problem.

1. Inspired by the optimal hyperparameter values obtained in D.2.6, do a search with 2-fold cross validation. Even though the default loss function in XGBoost is `squarederror`, hyperparameter values similar to the optimal hyperparameter values obtained in D.2.6 seem to work well. The regularization parameters `gamma` and `reg_lambda` help reduce MAE further.
2. It took 10 minutes on a 6-core laptop to tune the model with (1), with the values of `gamma` considered as 0 and 10, and values of `reg_lambda` considered as 0, 1, and 10.

(14 points)

D.2.11 XGBoost Feature importance

Arrange and print the predictors in decreasing order of importance.

(2 points)

D.3 Classification - Term deposit

The data for this question is related with direct marketing campaigns of a Portuguese banking institution. The marketing campaigns were based on phone calls, where bank clients were called to subscribe for a term deposit.

There is a train data - *train.csv*, which you will use to develop a model. There is a test data - *test.csv*, which you will use to test your model. Each dataset has the following attributes about the clients called in the marketing campaign:

1. `age`: Age of the client
2. `education`: Education level of the client
3. `day`: Day of the month the call is made
4. `month`: Month of the call
5. `y`: did the client subscribe to a term deposit?

6. `duration`: Call duration, in seconds. This attribute highly affects the output target (e.g., if `duration`=0 then `y`='no'). Yet, the duration is not known before a call is performed. Also, after the end of the call `y` is obviously known. Thus, this input should only be included for inference purposes and should be discarded if the intention is to have a realistic predictive model.

(Raw data source: Source. Do not use the raw data source for this assignment. It is just for reference.)

D.3.1 Data preparation

Convert all the categorical predictors in the data to dummy variables. Note that `month` and `education` are categorical variables.

(1 point)

D.3.2 Boosting

Develop and tune any boosting model to predict the probability of a client subscribing to a term deposit based on `age`, `education`, `day` and `month`. The model must have:

(a) **Minimum overall classification accuracy of 70%** among the classification accuracies on *train.csv*, and *test.csv*.

(b) **Minimum recall of 65%** among the recall on *train.csv*, and *test.csv*.

Print the accuracy and recall for both the datasets - *train.csv*, and *test.csv*.

Note that:

i. You cannot use `duration` as a predictor. The predictor is not useful for prediction because its value is determined after the marketing call ends. However, after the call ends, we already know whether the client responded positively or negatively.

ii. You are free to choose any value of threshold probability for classifying observations. However, you must use the same threshold on both the datasets.

iii. Use cross-validation on train data to optimize the model hyperparameters.

iv. Using the optimal model hyperparameters obtained in (iii), develop the boosting model. Plot the cross-validated accuracy and recall against decision threshold probability. Tune the decision threshold probability based on the plot, or the data underlying the plot to achieve the required trade-off between recall and accuracy.

v. Evaluate the accuracy and recall of the developed model with the tuned decision threshold probability on both the datasets. Note that the test dataset must only be used to evaluate performance metrics, and not optimize any hyperparameters or decision threshold probability.

(20 points - 10 points for tuning the hyperparameters, 4 points for making the plot, 4 points for tuning the decision threshold probability based on the plot, and 2 points for printing the accuracy & recall on both the datasets)

It is up to you to pick the hyperparameters and their values in the grid.

Hint: Below is one way to solve the problem. Note that there may be several completely different and better ways to solve the problem.

XGBoost may help with tuning of `n_estimators`, `max_depth`, `learning_rate`, `gamma`, `reg_lambda`, and `subsample`. You may take the recommended value of `scale_pos_weight`. Use `RandomizedSearchCV`. Evaluation of 200 models with 5-fold cross validation, i.e., 1000 fits, takes 45 minutes on a 6-core laptop. You may try a 2-fold cross validation to reduce time.

E Stratified splitting (classification problem)

E.1 Stratified splitting with respect to response

Q: When splitting data into train and test for developing and assessing a classification model, it is recommended to stratify the split with respect to the response. Why?

A: The main advantage of stratified splitting is that it can help ensure that the training and testing sets have similar distributions of the target variable, which can lead to more accurate and reliable model performance estimates.

In many real-world datasets, the target variable may be imbalanced, meaning that one class is more prevalent than the other(s). For example, in a medical dataset, the majority of patients may not have a particular disease, while only a small fraction may have the disease. If a random split is used to divide the dataset into training and testing sets, there is a risk that the testing set may not have enough samples from the minority class, which can lead to biased model performance estimates.

Stratified splitting addresses this issue by ensuring that both the training and testing sets have similar proportions of the target variable. This can lead to more accurate model performance estimates, especially for imbalanced datasets, by ensuring that the testing set contains enough samples from each class to make reliable predictions.

Another advantage of stratified splitting is that it can help ensure that the model is not overfitting to a particular class. If a random split is used and one class is overrepresented in the training set, the model may learn to predict that class well but perform poorly on the other class(es). Stratified splitting can help ensure that the model is exposed to a representative sample of all classes during training, which can improve its generalization performance on new, unseen data.

In summary, the advantages of stratified splitting are that it can lead to more accurate and reliable model performance estimates, especially for imbalanced datasets, and can help prevent overfitting to a particular class.

E.2 Stratified splitting with respect to response and categorical predictors

Q: Will it be better to stratify the split with respect to the response as well as categorical predictors, instead of only the response? In that case, the train and test datasets will be even more representative of the complete data.

A: It is not recommended to stratify with respect to both the response and categorical predictors simultaneously, while splitting a dataset into train and test, because doing so may result in the test data being very similar to train data, thereby defeating the purpose of assessing the model on unseen data. This kind of a stratified splitting will tend to make the relationships between the response and predictors in train data also appear in test data, which will result in the performance on test data being very similar to that in train data. Thus, in this case, the ability of the model to generalize to new, unseen data won't be assessed by test data.

Therefore, it is generally recommended to only stratify the response variable when splitting the data for model training, and to use random sampling for the predictor variables. This helps to ensure that the model is able to capture the underlying relationships between the predictor variables and the response variable, while still being able to generalize well to new, unseen data.

In the extreme scenario, when there are no continuous predictors, and there are enough observations for stratification with respect to the response and the categorical predictors, the train and test datasets may turn out to be exactly the same. Example 1 below illustrates this scenario.

E.3 Example 1

The example below shows that the train and test data can be exactly the same if we stratify the split with respect to response and the categorical predictors.

```
# 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 LogisticRegression
from sklearn.model_selection import train_test_split, cross_val_predict, cross_val_score
from sklearn.metrics import accuracy_score
from itertools import product
sns.set(font_scale=1.35)
```

Let us simulate a dataset with 8 observations, two categorical predictors `x1` and `x2` and the the binary response `y`.

```
#Setting a seed for reproducible results
np.random.seed(9)

# 8 observations
n = 8

#Simulating the categorical predictors
x1 = pd.Series(np.random.randint(0,2,n), name = 'x1')
x2 = pd.Series(np.random.randint(0,2,n), name = 'x2')

#Simulating the response
pr = (x1==1)*0.7+(x2==0)*0.3# + (x3*0.1>0.1)*0.1
y = pd.Series(1*(np.random.uniform(size = n) < pr), name = 'y')

#Defining the predictor object 'X'
X = pd.concat([x1, x2], axis = 1)

#Stratified splitting with respect to the response and predictors to create 50% train and
X_train_stratified, X_test_stratified, y_train_stratified,\
y_test_stratified = train_test_split(X, y, test_size = 0.5, random_state = 45, stratify=da

#Train and test data resulting from the above stratified splitting
data_train = pd.concat([X_train_stratified, y_train_stratified], axis = 1)
data_test = pd.concat([X_test_stratified, y_test_stratified], axis = 1)
```

Let us check the train and test datasets created with stratified splitting with respect to both the predictors and the response.

```
data_train
```

	x1	x2	y
2	0	0	1
7	0	1	0
3	1	0	1
1	0	1	0

```
data_test
```

	x1	x2	y
4	0	1	0
6	1	0	1
0	0	1	0
5	0	0	1

Note that the train and test datasets are exactly the same! Stratified splitting tends to have the same proportion of observations corresponding to each strata in both the train and test datasets, where each strata is a unique combination of values of `x1`, `x2`, and `y`. This will tend to make the train and test datasets quite similar!

E.4 Example 2: Simulation results

The example below shows that train and test set performance will tend to be quite similar if we stratify the datasets with respect to the predictors and the response.

We'll simulate a dataset consisting of 1000 observations, 2 categorical predictors `x1` and `x2`, a continuous predictor `x3`, and a binary response `y`.

```
#Setting a seed for reproducible results
np.random.seed(99)

# 1000 Observations
n = 1000

#Simulating categorical predictors x1 and x2
x1 = pd.Series(np.random.randint(0,2,n), name = 'x1')
x2 = pd.Series(np.random.randint(0,2,n), name = 'x2')

#Simulating continuous predictor x3
x3 = pd.Series(np.random.normal(0,1,n), name = 'x3')

#Simulating the response
pr = (x1==1)*0.7+(x2==0)*0.3 + (x3*0.1>0.1)*0.1
y = pd.Series(1*(np.random.uniform(size = n) < pr), name = 'y')

#Defining the predictor object 'X'
X = pd.concat([x1, x2, x3], axis = 1)
```

We'll comparing model performance metrics when the data is split into train and test by performing stratified splitting

1. Only with respect to the response
2. With respect to the response and categorical predictors

We'll perform 1000 simulations, where the data is split using a different seed in each simulation.

```
#Creating an empty dataframe to store simulation results of 1000 simulations
accuracy_iter = pd.DataFrame(columns = {'train_y_stratified','test_y_stratified',
                                        'train_y_CatPredictors_stratified','test_y_CatPred
```

```
# Comparing model performance metrics when the data is split into train and test by perfor
# (1) only with respect to the response
# (2) with respect to the response and categorical predictors

# Stratified splitting is performed 1000 times and the results are compared
for i in np.arange(1,1000):

    #--------Case 1-------------------#
    # Stratified splitting with respect to response only to create train and test data
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2, random_stat
    model = LogisticRegression()
    model.fit(X_train, y_train)

    # Model accuracy on train and test data, with stratification only on response while sp
    # the complete data into train and test
    accuracy_iter.loc[(i-1), 'train_y_stratified'] = model.score(X_train, y_train)
    accuracy_iter.loc[(i-1), 'test_y_stratified'] = model.score(X_test, y_test)

    #--------Case 2-------------------#
    # Stratified splitting with respect to response and categorical predictors to create t
    # and test data
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2, random_stat
                                                        stratify=pd.concat([x1, x2, y], ax
    model.fit(X_train, y_train)

    # Model accuracy on train and test data, with stratification on response and predictor
    # splitting the complete data into train and test
    accuracy_iter.loc[(i-1), 'train_y_CatPredictors_stratified'] = model.score(X_train, y_
    accuracy_iter.loc[(i-1), 'test_y_CatPredictors_stratified'] = model.score(X_test, y_te

# Converting accuracy to numeric
```

```
accuracy_iter = accuracy_iter.apply(lambda x:x.astype(float), axis = 1)
```

Distribution of train and test accuracies

The table below shows the distribution of train and test accuracies when the data is split into train and test by performing stratified splitting:

1. Only with respect to the response (see `train_y_stratified` and `test_y_stratified`)
2. With respect to the response and categorical predictors (see `train_y_CatPredictors_stratified` and `test_y_CatPredictors_stratified`)

```
accuracy_iter.describe()
```

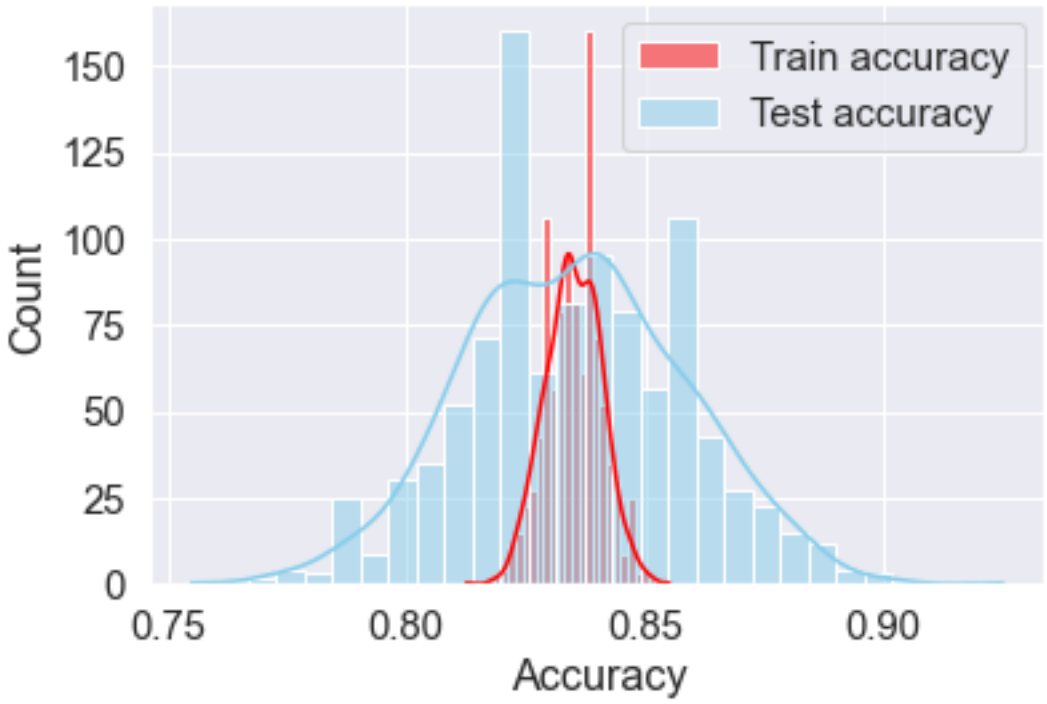
	train_y_stratified	test_y_stratified	train_y_CatPredictors_stratified	test_y_CatPredictors_st
count	999.000000	999.000000	9.990000e+02	9.990000e+02
mean	0.834962	0.835150	8.350000e-01	8.350000e-01
std	0.005833	0.023333	8.552999e-15	8.552999e-15
min	0.812500	0.755000	8.350000e-01	8.350000e-01
25%	0.831250	0.820000	8.350000e-01	8.350000e-01
50%	0.835000	0.835000	8.350000e-01	8.350000e-01
75%	0.838750	0.850000	8.350000e-01	8.350000e-01
max	0.855000	0.925000	8.350000e-01	8.350000e-01

Let us visualize the distribution of these accuracies.

E.4.1 Stratified splitting only with respect to the response

```
sns.histplot(data=accuracy_iter, x="train_y_stratified", color="red", label="Train accurac
sns.histplot(data=accuracy_iter, x="test_y_stratified", color="skyblue", label="Test accur
plt.legend()
plt.xlabel('Accuracy')
```

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

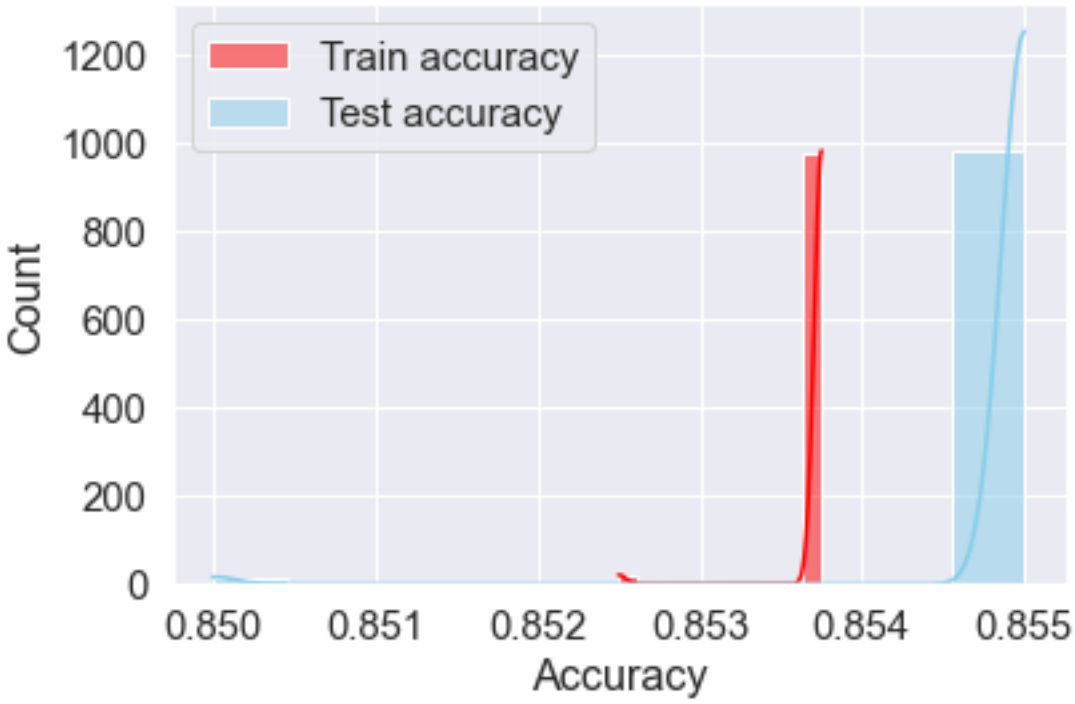


Note the variability in train and test accuracies when the data is stratified only with respect to the response. The train accuracy varies between 81.2% and 85.5%, while the test accuracy varies between 75.5% and 92.5%.

E.4.2 Stratified splitting with respect to the response and categorical predictors

```
sns.histplot(data=accuracy_iter, x="train_y_CatPredictors_stratified", color="red", label=
sns.histplot(data=accuracy_iter, x="test_y_CatPredictors_stratified", color="skyblue", lab
plt.legend()
plt.xlabel('Accuracy')
```

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



The train and test accuracies are between 85% and 85.5% for all the simulations. As a results of stratifying the splitting with respect to both the response and the categorical predictors, the train and test datasets are almost the same because the datasets are engineered to be quite similar, thereby making the test dataset inappropriate for assessing accuracy on unseen data. Thus, it is recommended to stratify the splitting only with respect to the response.

F GridSearchCV vs RandomSearchCV

Which is better - `GridSearchCV` or `RandomizedSearchCV`?

Let us understand with the help of scenarios.

F.1 Infinite budget

Suppose we have infinite computational budget. This is practically possible if the dataset set size is small, i.e., the number of observations and the number of predictors are small.

In case of infinite budget, we can evaluate the cross-validated model score on all potential values of hyperparameters. Thus, in this case `GridSearchCV` is the better choice, as it won't leave out any possibility.

F.2 Finite budget

Generally, we always have a finite compuational budget, and can perform a certain maximum number of model evaluations. However, let's consider 2 cases:

F.2.1 Large number of hyperparameters

If we have a large number of hyperparameters, say H and want to compute the cross validation score for at least 2 distinct values for each hyperparameter, then the number of models to be fit with `GridSearchCV` will be 2^H. In this case, we'll certainly need to use `RandomizedSearchCV` for a large H.

F.2.2 Small number of hyperparameters

Let us consider the case when we have a small number of hyperparameters, say 2. In this case which one will be better - `GridSearchCV` or `RandomizedSearchCV`? Let us consider an example.

```
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, RandomizedSearchCV
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_sc
accuracy_score, precision_score, confusion_matrix, mean_squared_error, r2_score

#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
```

```
#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']
```

F.2.3 Example

Suppose we wish to optimize 2 hyperparameters - `max_samples` and `max_depth` for a bagged trees model that predicts car price. We wish to consider `max_depth` $\in \{2, 4, 6\}$, and `max_samples` $\in [0.4, 1.0]$. Suppose we have a computational budget of testing 9 combinations of hyperparameter values.

Which approach will be better?

F.2.3.1 `GridSearchCV`

With `GridSearchCV`, let us consider `max_samples` $\in \{0.4, 0.7, 1.0\}$ as we have a budget of testing 9 combinations of hyperparameter values, and we are already considering `max_depth` $\in \{2, 4, 6\}$

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

params = {'base_estimator__max_depth':[2, 4, 6],
          'max_samples': [0.4, 0.7, 1.0]}

cv = KFold(n_splits=5,shuffle=True,random_state=1)
bagging_regressor_grid = GridSearchCV(BaggingRegressor(base_estimator=DecisionTreeRegresso
                                                       random_state=1, n_jobs=-1, n_estima
                                                      bootstrap = False),
                                      param_grid =params, cv=cv, n_jobs=-1, verbose=1,
                                     scoring='neg_root_mean_squared_error')
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 RMSE through Grid Search : %.3f'%-bagging_regressor_grid.best_score_)
print('Best Parameters : ',bagging_regressor_grid.best_params_)
```

```
Fitting 5 folds for each of 9 candidates, totalling 45 fits
```

```
Train R^2 Score : 0.886
Test R^2 Score : 0.839
Best RMSE through Grid Search : 6304.919
Best Parameters :  {'base_estimator__max_depth': 6, 'max_samples': 0.7}
```

We get a cross-validated RMSE of $6304.92 with `GridSearchCV`.

```
#GridSearchCV results
cv_results = pd.DataFrame(bagging_regressor_grid.cv_results_)
```

F.2.3.2 `RandomizedSearchCV`

With `RandomizedSearchCV`, we can consider as many distinct values of `max_samples` as we wish, where `max_samples` $\in [0.4, 1.0]$, since we control the number of combinations of hyperparameter values (`n_iter`) for which the cross validation score will be computed. Let us consider `max_samples` $\in [0.4, 0.47, 0.54, 0.61, 0.68, 0.75, 0.82, 0.89, 0.96]$

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

params = {'base_estimator__max_depth':[2, 4, 6],
          'max_samples': np.arange(0.4,1.01,0.07)}

cv = KFold(n_splits=5,shuffle=True,random_state=1)
bagging_regressor_rgrid = RandomizedSearchCV(BaggingRegressor(base_estimator=DecisionTreeR
                                                              random_state=1, n_jobs=-1, n_
                                                              bootstrap = False),
                                             param_distributions =params, cv=cv, n_jobs=-1, verbo
                                                  random_state=1, scoring='neg_root_mean_squared_
bagging_regressor_rgrid.fit(X, y)

print('Train R^2 Score : %.3f'%bagging_regressor_rgrid.best_estimator_.score(X, y))
print('Test R^2 Score : %.3f'%bagging_regressor_rgrid.best_estimator_.score(Xtest, ytest))
print('Best RMSE through random Search : %.3f'%-bagging_regressor_rgrid.best_score_)
print('Best Parameters : ',bagging_regressor_rgrid.best_params_)
```

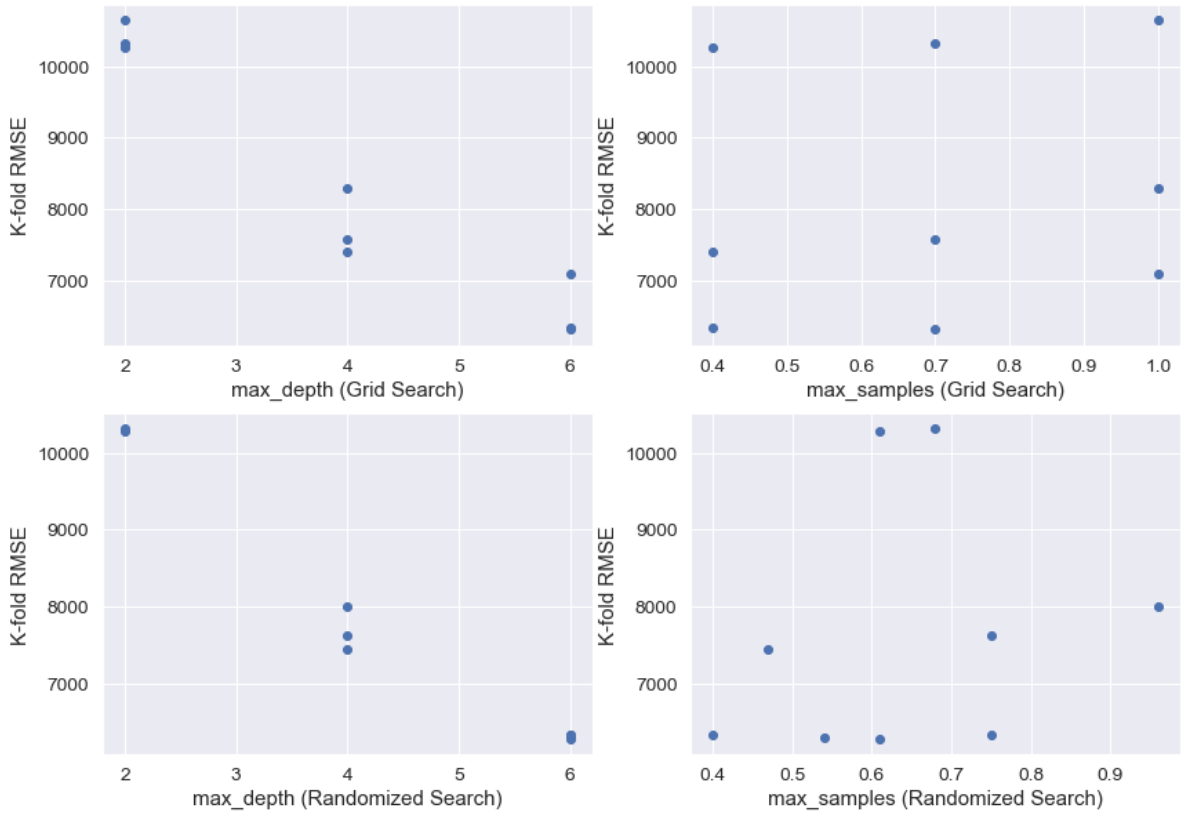
```
Fitting 5 folds for each of 9 candidates, totalling 45 fits
Train R^2 Score : 0.885
Test R^2 Score : 0.840
Best RMSE through random Search : 6275.076
Best Parameters :  {'max_samples': 0.6100000000000001, 'base_estimator__max_depth': 6}
```

We get a cross-validated RMSE of $6275.08 with `RandomizedSearchCV`, which is better than what we got with `GridSearchCV`. Although not gauranteed, `RandomizedSearchCV` is likely to do better than `GridSearchCV` even for a few hyperparameter values.

Let us understand this by analyzing the cross validated results of both types of searches.

```
#RandomizedSearchCV results
cv_resultsr = pd.DataFrame(bagging_regressor_rgrid.cv_results_)
```

```
plt.rcParams["figure.figsize"] = (9,6)
sns.set(font_scale=1.25)

#GridSearchCV
fig, axes = plt.subplots(2,2,figsize=(14,10))
plt.subplots_adjust(wspace=0.2)
axes[0, 0].plot(cv_results.param_base_estimator__max_depth, (-cv_results.mean_test_score),
axes[0, 0].set_xlabel('max_depth (Grid Search)')
axes[0, 0].set_ylabel('K-fold RMSE')
axes[0, 1].plot(cv_results.param_max_samples, (-cv_results.mean_test_score), 'o')
axes[0, 1].set_xlabel('max_samples (Grid Search)')
axes[0, 1].set_ylabel('K-fold RMSE')

#RandomizedSearchCV
axes[1, 0].plot(cv_resultsr.param_base_estimator__max_depth, (-cv_resultsr.mean_test_score
axes[1, 0].set_xlabel('max_depth (Randomized Search)')
axes[1, 0].set_ylabel('K-fold RMSE')
axes[1, 1].plot(cv_resultsr.param_max_samples, (-cv_resultsr.mean_test_score), 'o')
axes[1, 1].set_xlabel('max_samples (Randomized Search)')
axes[1, 1].set_ylabel('K-fold RMSE');
```



Note that the trend for `max_depth` appears to be similar for both the searches regardless of the value of `max_samples`. However, with `RandomizedSearchCV`, we are able to compute the cross validation score for more distinct values of `max_samples`. We are obtaining low cross-validated RMSEs for `max_samples` values of 0.54, 0.61, and 0.75, which we miss out in `GridSearchCV`.

`GridSearchCV`, only searches 3 different values of `max_samples` in 9 iterations. However, `RandomizedSearchCV` can search 9 different values for the 9 iterations. As a result, it is much easier for `RandomizedSearchCV` to search for the optimal hyperparameters, and is thus more likely to find more optimal hyperparameter values.

Let us compute the RMSE on test data for the optimal models obtained with both types of search.

```
#RMSE for the optimal hyperparameters based on GridSearchCV
np.sqrt(mean_squared_error(test.price, bagging_regressor_grid.predict(Xtest)))
```

```
6661.573791208074
```

```
#RMSE for the optimal hyperparameters based on RandomizedSearchCV
np.sqrt(mean_squared_error(test.price, bagging_regressor_rgrid.predict(Xtest)))
```

```
6640.462268475577
```

We oberve that `RandomizedSearchCV` seems to have found slightly better optimal hyperparameter values than `GridSearchCV` for the same number of model fits. Though likely, this is not gauranteed to happen.

G Datasets, assignment and project files

Datasets used in the book, assignment files, project files, and prediction problems report tempate can be found here

References