

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

**STAT 303-3**

Arvind Krishna

2023-03-24

# Table of contents

<b>Preface</b>	<b>4</b>
<b>1 Introduction to scikit-learn</b>	<b>5</b>
1.1 Splitting data into <code>train</code> and <code>test</code> . . . . .	6
1.1.1 Stratified splitting . . . . .	7
1.2 Scaling data . . . . .	8
1.3 Fitting a model . . . . .	9
1.4 Computing performance metrics . . . . .	10
1.4.1 Accuracy . . . . .	10
1.4.2 ROC-AUC . . . . .	11
1.4.3 Confusion matrix & precision-recall . . . . .	11
1.5 Tuning the model hyperparameters . . . . .	14
1.5.1 Tuning decision threshold probability . . . . .	16
1.5.2 Tuning the regularization parameter . . . . .	19
1.5.3 Tuning the decision threshold probability and the regularization parameter simultaneously . . . . .	22
<b>2 Bias-variance tradeoff</b>	<b>26</b>
2.1 Simple model (Less flexible) . . . . .	26
2.2 Complex model (more flexible) . . . . .	29
<b>3 Regression trees</b>	<b>32</b>
3.1 Building a regression tree . . . . .	33
3.2 Optimizing parameters to improve the regression tree . . . . .	35
3.2.1 Range of hyperparameter values . . . . .	35
3.2.2 Cross validation: Coarse grid . . . . .	36
3.2.3 Cross validation: Finer grid . . . . .	37
3.3 Cost complexity pruning . . . . .	39
3.3.1 Depth vs alpha; Node counts vs alpha . . . . .	42
3.3.2 Train and test accuracies (R-squared) vs alpha . . . . .	43
<b>4 Classification trees</b>	<b>45</b>
4.1 Building a classification tree . . . . .	46
4.2 Optimizing hyperparameters to optimize performance . . . . .	48
4.3 Optimizing the decision threshold probability . . . . .	50
4.3.1 Balancing recall with precision . . . . .	50

4.3.2	Balancing recall with false positive rate . . . . .	55
4.4	Cost complexity pruning . . . . .	59

## **Appendices** **61**

### **A Stratified splitting (classification problem)** **61**

A.1	Stratified splitting with respect to response . . . . .	61
A.2	Stratified splitting with respect to response and categorical predictors . . . . .	62
A.3	Example 1 . . . . .	62
A.4	Example 2: Simulation results . . . . .	64
	Distribution of train and test accuracies . . . . .	66
A.4.1	Stratified splitting only with respect to the response . . . . .	66
A.4.2	Stratified splitting with respect to the response and categorical predictors	67

### **B Datasets, assignment and project files** **69**

# 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!

# 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  $\lambda$ , 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 validation
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_score, 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 (different ways)
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 is large
# 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 = 45)
```

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 different
# of magn., scaling is necessary. This is to enable the training algo. which we did not cover
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, just transform
```



## 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\AppData\Roaming\Python\Python38\site-packages\sklearn\linear_model\_logistic.py:1181:
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](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 (y=1)
#The 2nd column in the output of predict_proba() consists of the probability of the observation
#belong to the positive class (y=1)
y_pred_prob = logreg.predict_proba(X_test_scaled)[:,-1]

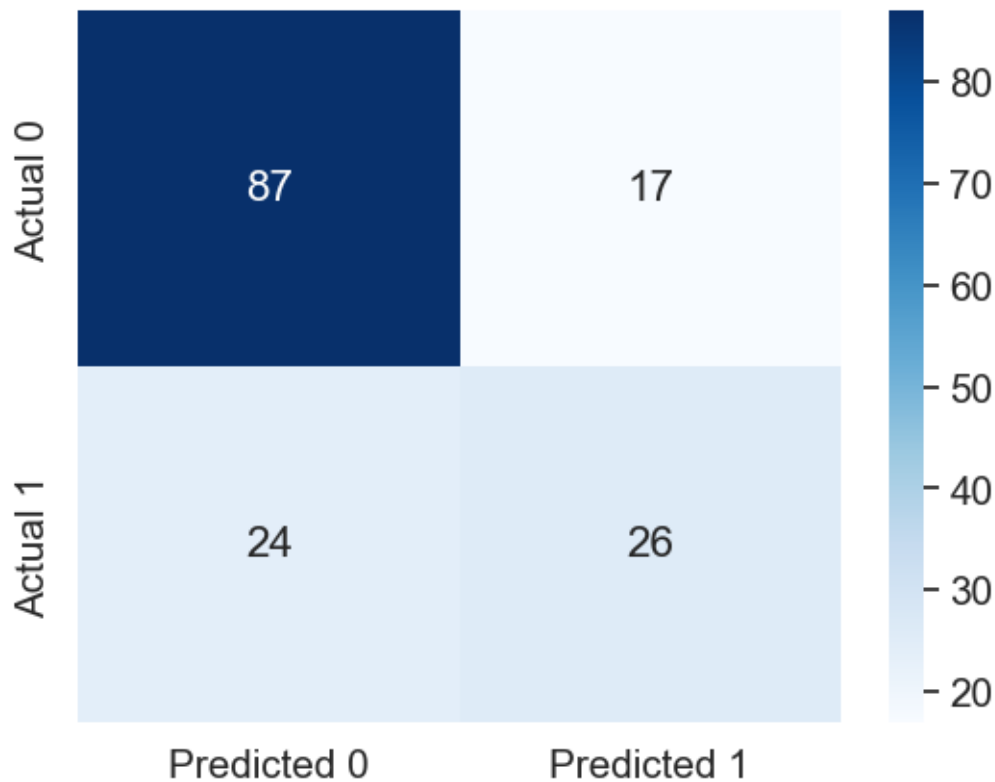
#Using the predicted probability computed above to find ROC-AUC
fpr, tpr, auc_thresholds = roc_curve(y_test, y_pred_prob)
print(auc(fpr, tpr))# AUC of ROC
```

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))

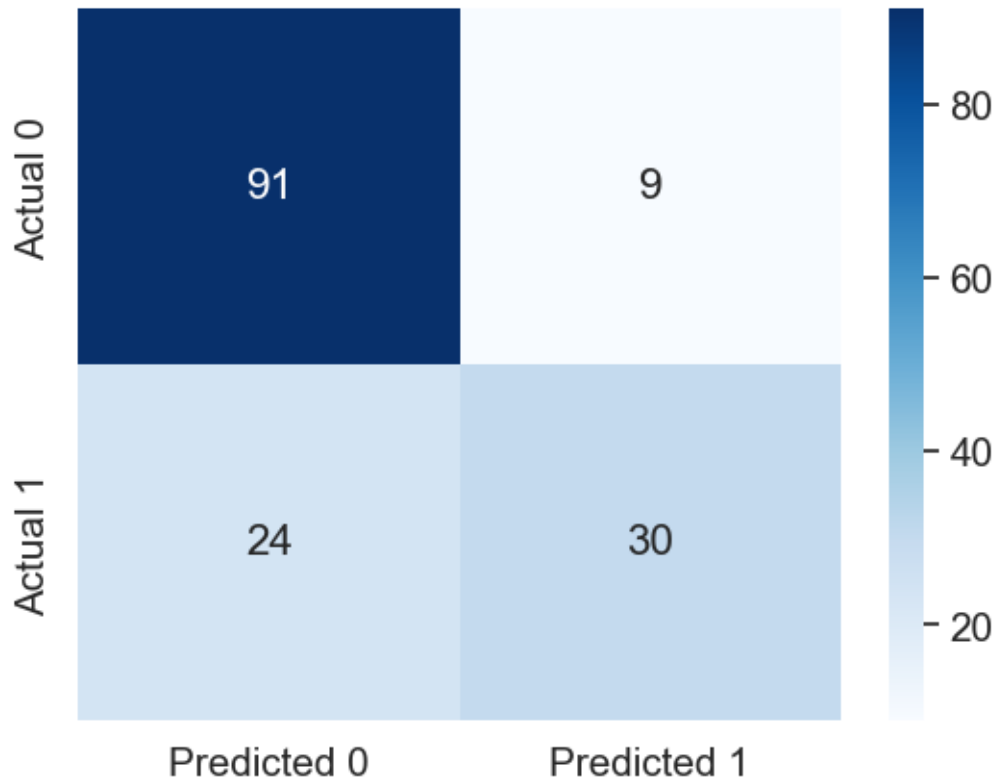
#Confusion matrix
cm = pd.DataFrame(confusion_matrix(y_test_stratified, y_pred_stratified), columns=['Predicted', 'Actual'],
                  index = ['Actual 0', 'Actual 1'])
sns.heatmap(cm, annot=True, cmap='Blues', fmt='g');

```

```

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 greater
# 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))

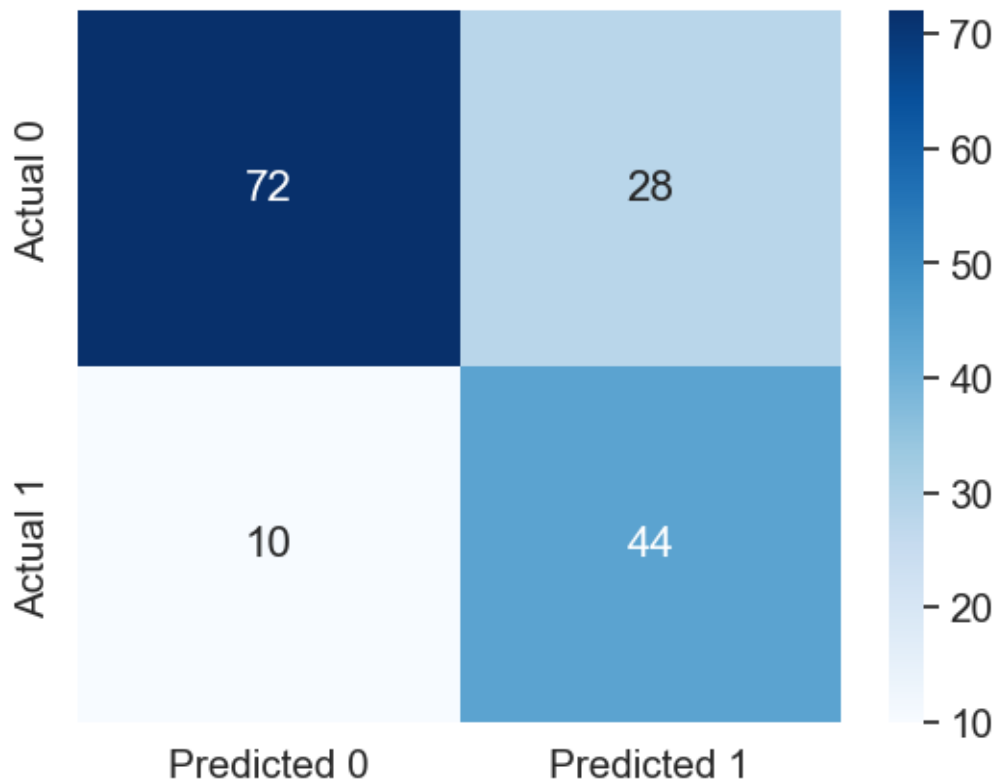
#Confusion matrix
cm = pd.DataFrame(confusion_matrix(y_test_stratified, y_pred_desired_threshold),
                  columns=['Predicted 0', 'Predicted 1'], index = ['Actual 0', 'Actual 1'])
sns.heatmap(cm, annot=True, cmap='Blues', fmt='g');

```

```

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 = 'predict_')
```



```

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)

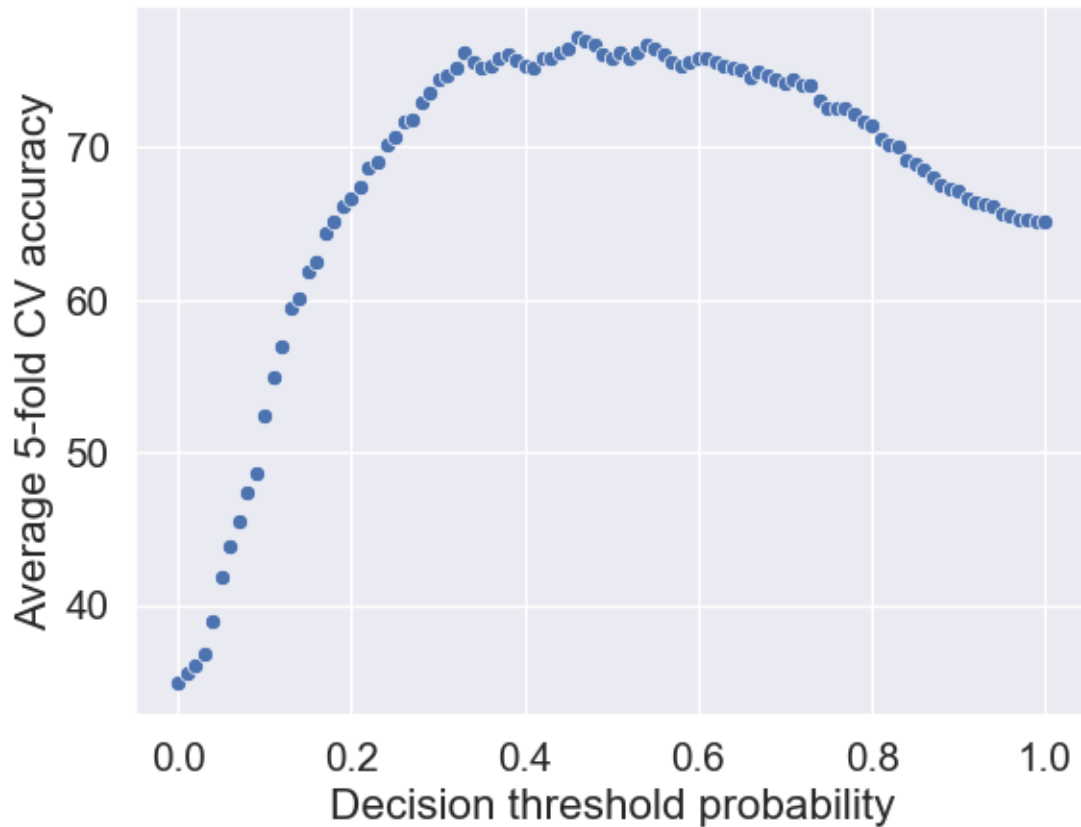
```

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 greater
# 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)

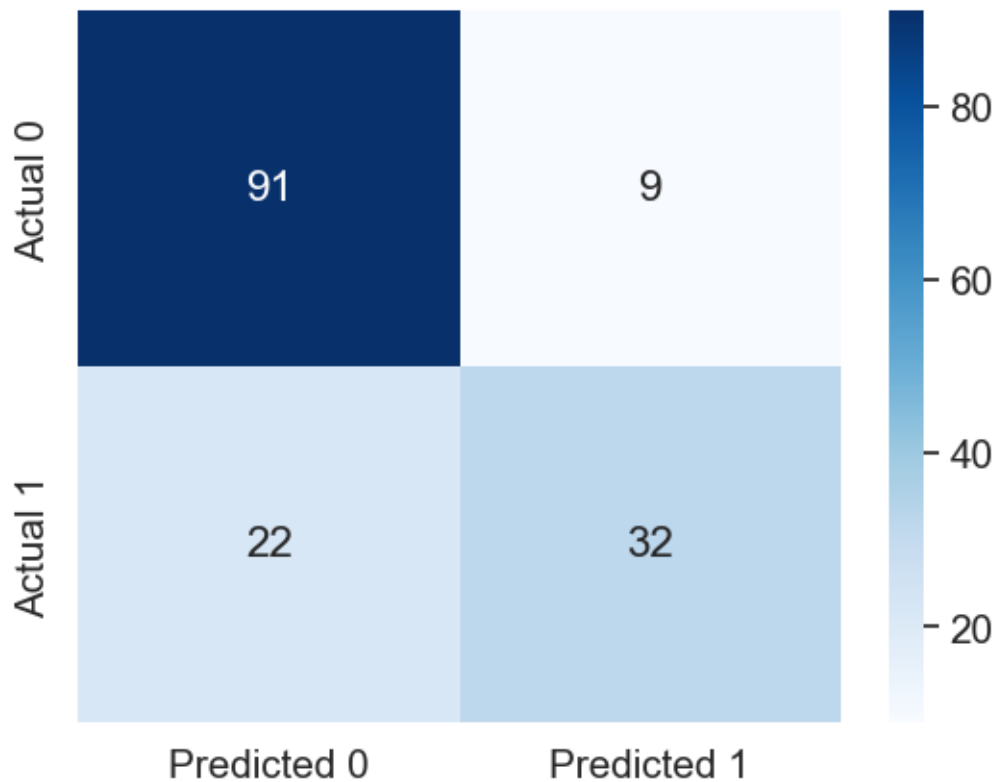
#Computing the accuracy
print("Accuracy: ",accuracy_score(y_pred_desired_threshold, y_test_stratified)*100)

#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

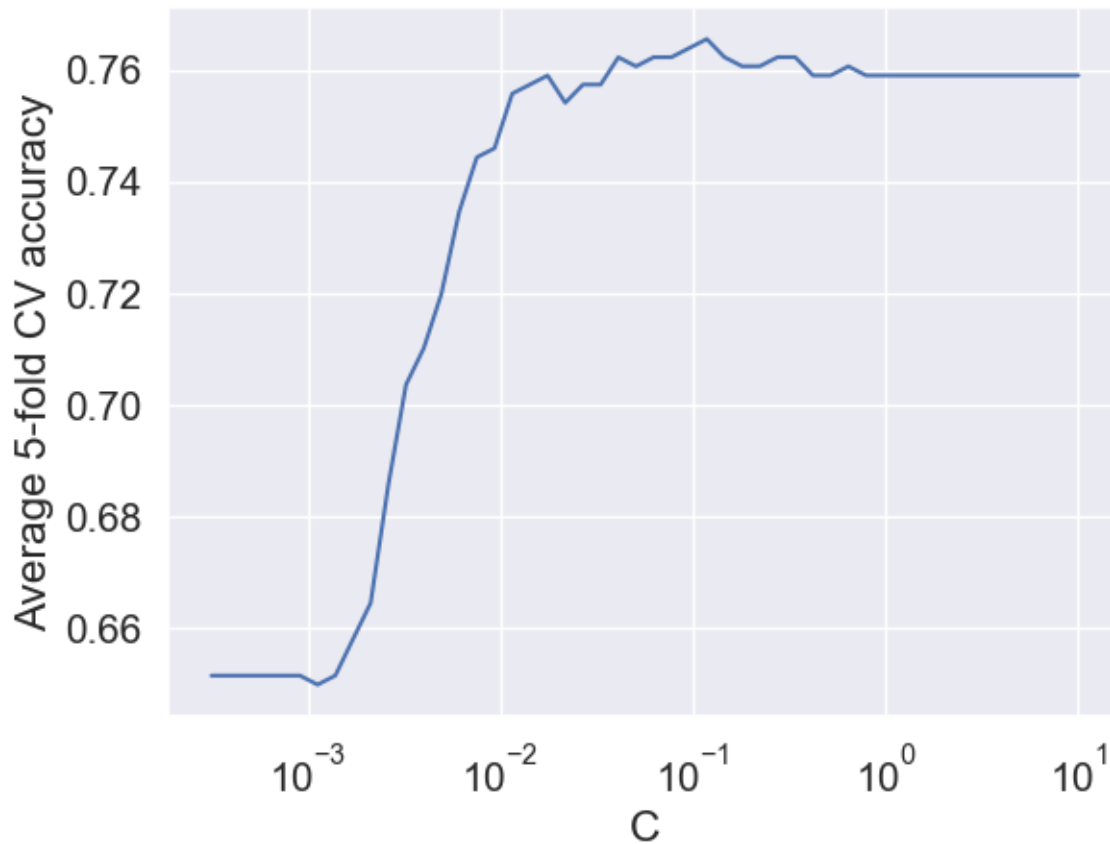
The `LogisticRegression()` method has a default  $L2$  regularization penalty, which means ridge regression.  $C$  is  $1/\lambda$ , where  $\lambda$  is the hyperparameter that is multiplied with the ridge penalty.  $C$  is 1 by default.

```
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_stratified_scaled,
                                         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))

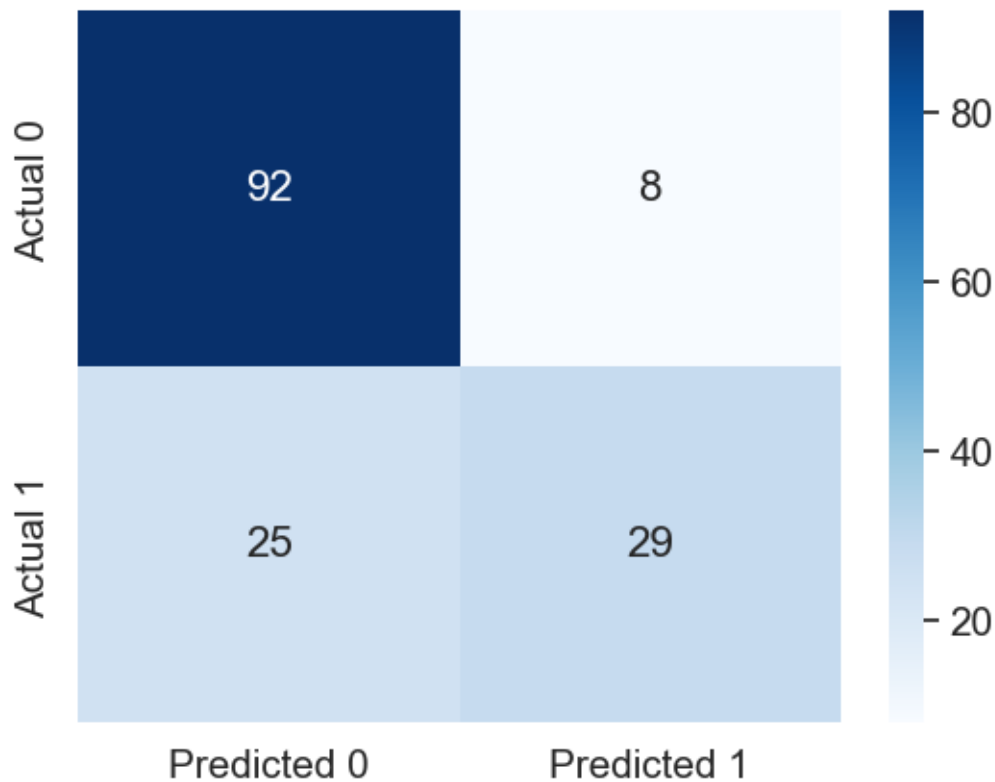
#Confusion matrix
cm = pd.DataFrame(confusion_matrix(y_test_stratified, y_pred_stratified), columns=['Predicted 0', 'Predicted 1'],
                  index = ['Actual 0', 'Actual 1'])
sns.heatmap(cm, annot=True, cmap='Blues', fmt='g');

```

```

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({'threshold':[], 'C':[], 'accuracy':[]})
iter_number = 0

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

    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, 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)

# 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 greater
# 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)

```

```

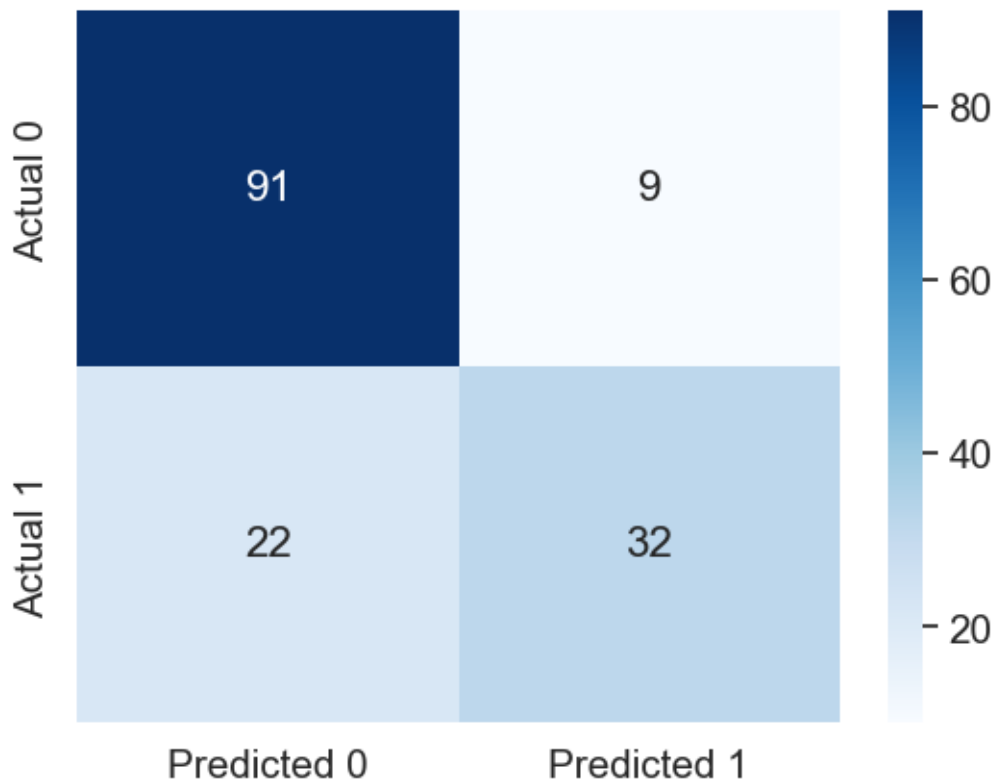
#Computing the ROC-AUC
fpr, tpr, auc_thresholds = roc_curve(y_test_stratified, y_pred_stratified_prob)
print("ROC-AUC: ",auc(fpr, tpr))# AUC of ROC

#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=['P', '1'])
cm.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 Bias-variance tradeoff

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

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

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

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

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

### 2.1 Simple model (Less flexible)

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

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

```
np.random.seed(101)

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

```

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

# Simulating test response by adding noise to the true mean response
ytest = fxtest + np.random.normal(0, 10, 200)

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

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

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

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

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

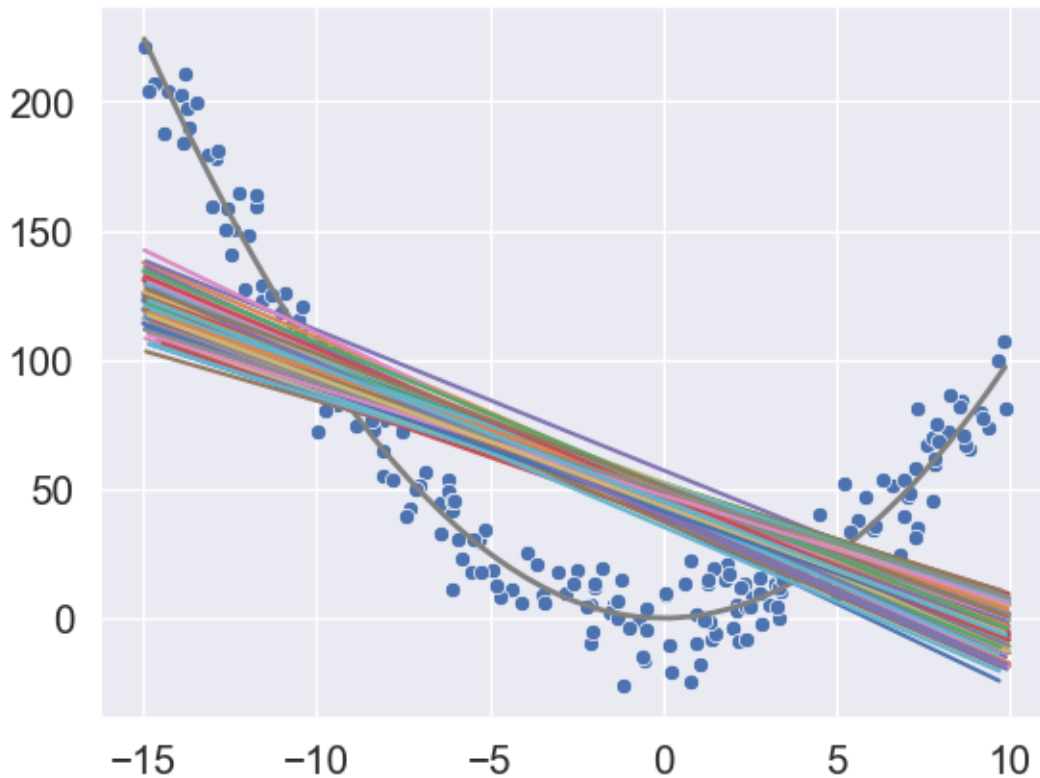
    # Fitting the ith model on the ith training data
    model.fit(x.reshape(-1,1), y)

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

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

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

```



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

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

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

2042.104126728109

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

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

28.37397844429763

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

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

2201.957555529835

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

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

```
sq_bias + mean_var + 100
```

2170.4781051724067

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

## 2.2 Complex model (more flexible)

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

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

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



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

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

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

1.3117561629333938

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

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

102.5226748977198

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

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

```
225.92027460924726
```

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

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

## 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

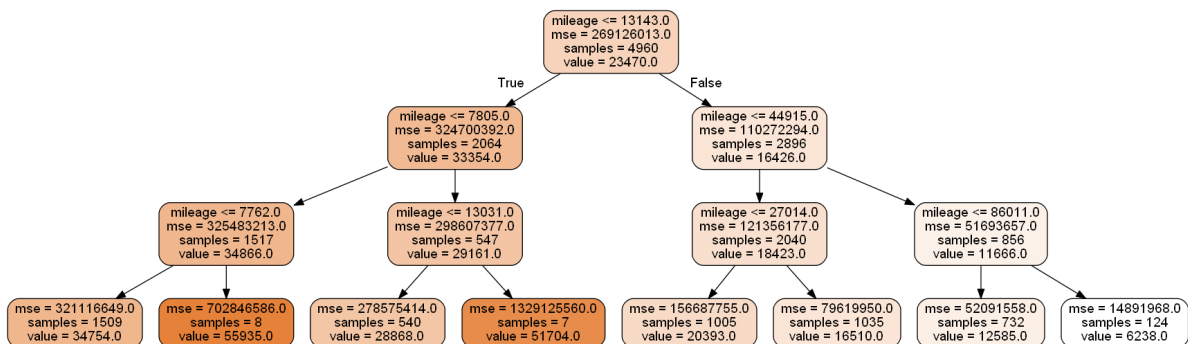
```
X = train['mileage']
y = train['price']
```

```
#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())
```

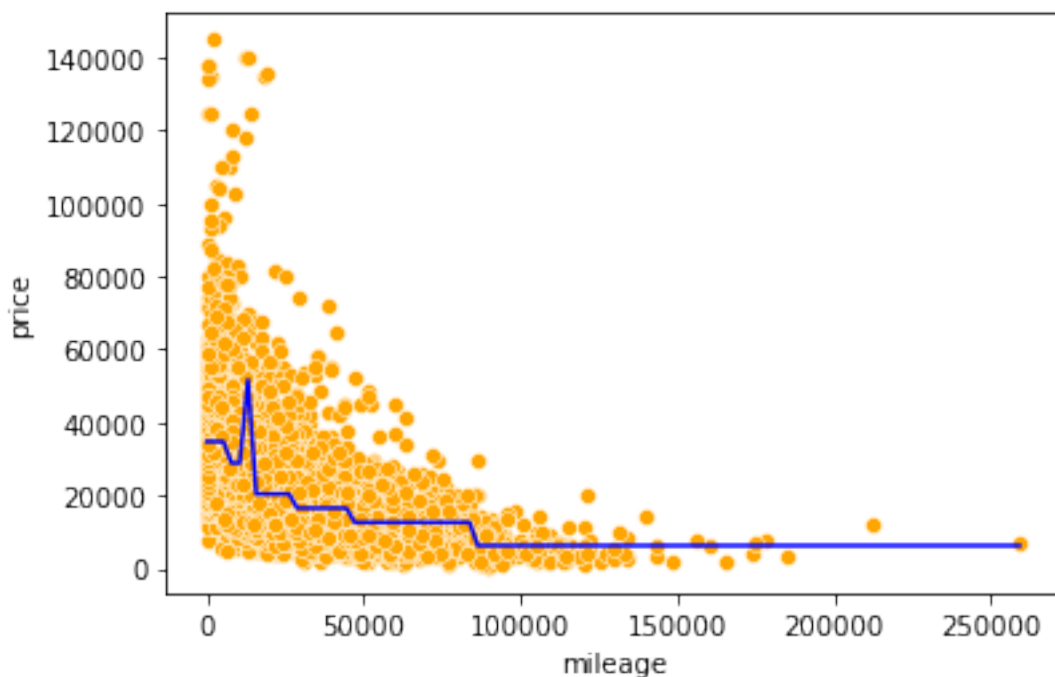


```
#prediction on test data
pred=model.predict(test[['mileage']])
```

```
#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')
```



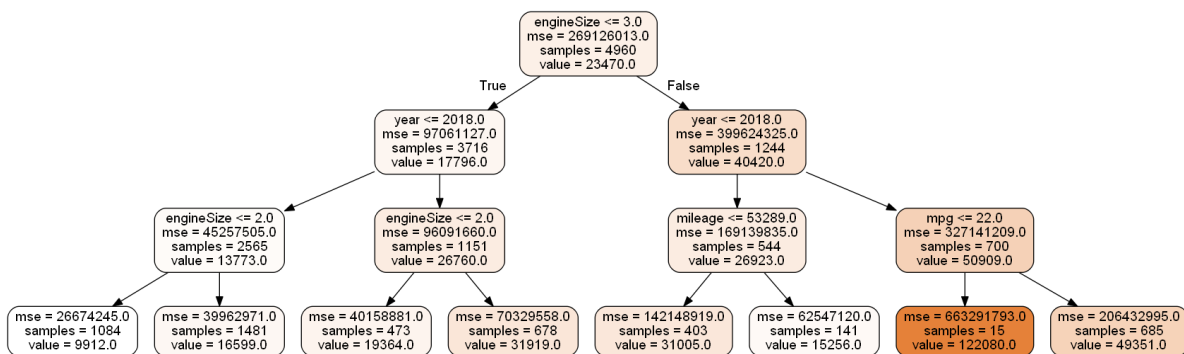
All cars falling within the same terminal node have the same predicted price, which is seen as flat line segments in the above model curve.

**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=1,
model.fit(X, y)
print (model.best_score_, model.best_params_)
```

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

Let us find the optimal hyperparameters based on 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=1,
                    scoring=['neg_mean_squared_error', 'r2'], refit = 'neg_mean_squared_error',
model.fit(X, y)
print (model.best_score_, model.best_params_)
```

Fitting 5 folds for each of 490 candidates, totalling 2450 fits  
-42064467.15261547 {'max\_depth': 11, 'max\_leaf\_nodes': 302}

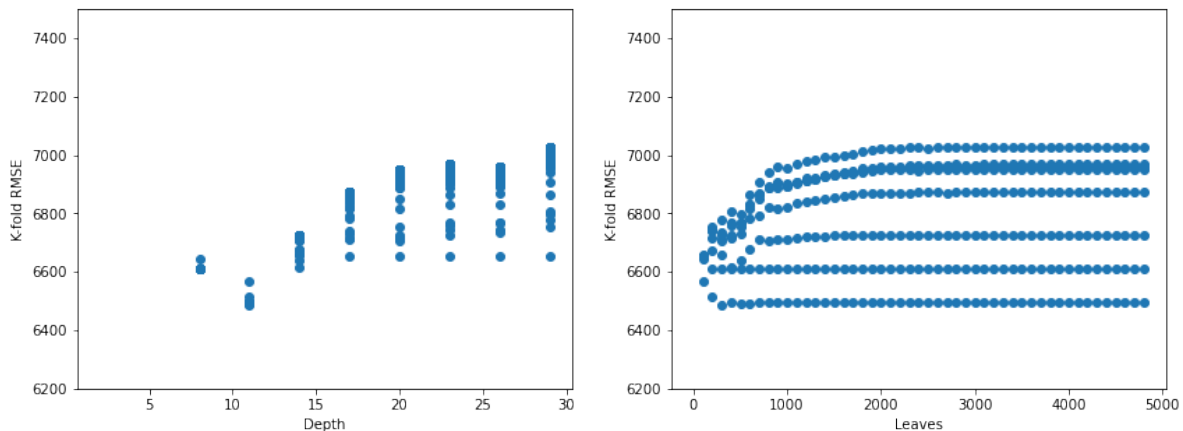
Note that as the `GridSearchCV` function maximizes the performance metric to find the optimal hyperparameters, we are maximizing the negative mean squared error

(`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_error))
axes[0].set_ylim([6200, 7500])
axes[0].set_xlabel('Depth')
axes[0].set_ylabel('K-fold RMSE')
axes[1].plot(cv_results.param_max_leaf_nodes, np.sqrt(-cv_results.mean_test_neg_mean_squared_error))
axes[1].set_ylim([6200, 7500])
axes[1].set_xlabel('Leaves')
axes[1].set_ylabel('K-fold RMSE');
```



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

### 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=1)
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.DecisionTreeRegressor.html>*

**Why?**

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

```
model.feature_importances_
```

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

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

### 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  $\alpha$ . All the non-terminal nodes for which  $\alpha_{eff}$  is smaller than the optimal  $\alpha$  will be pruned. You will need to check out the link below to understand this better.

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

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

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

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

```
len(alphas)
```

4126

```

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

```

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

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

```

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

```

7306.592294294368

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

```

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

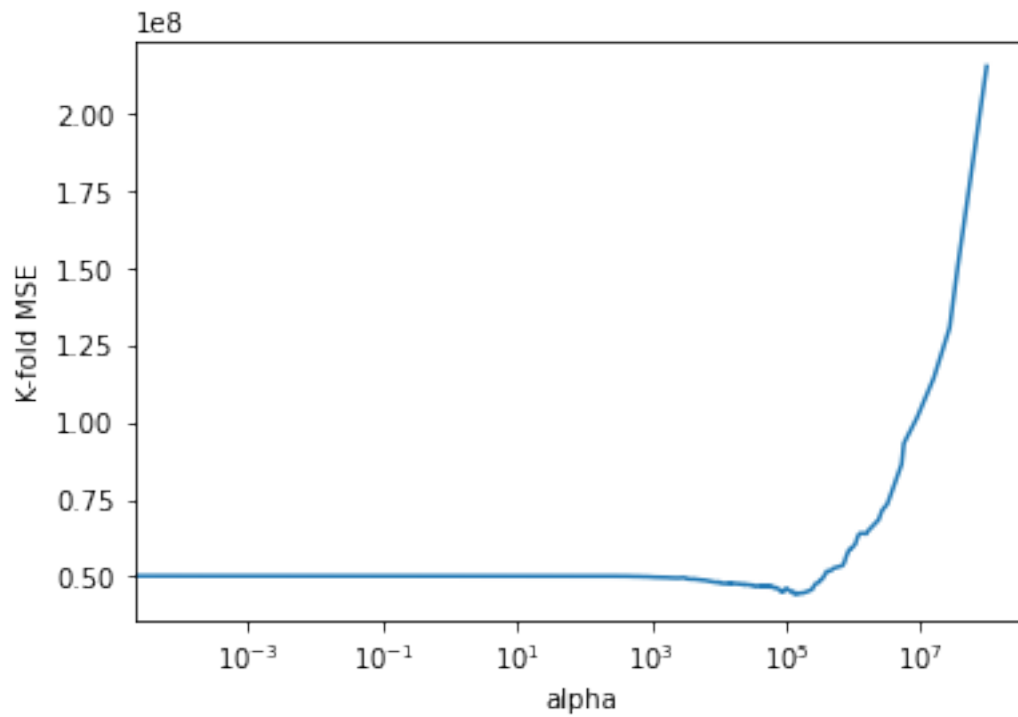
```

```

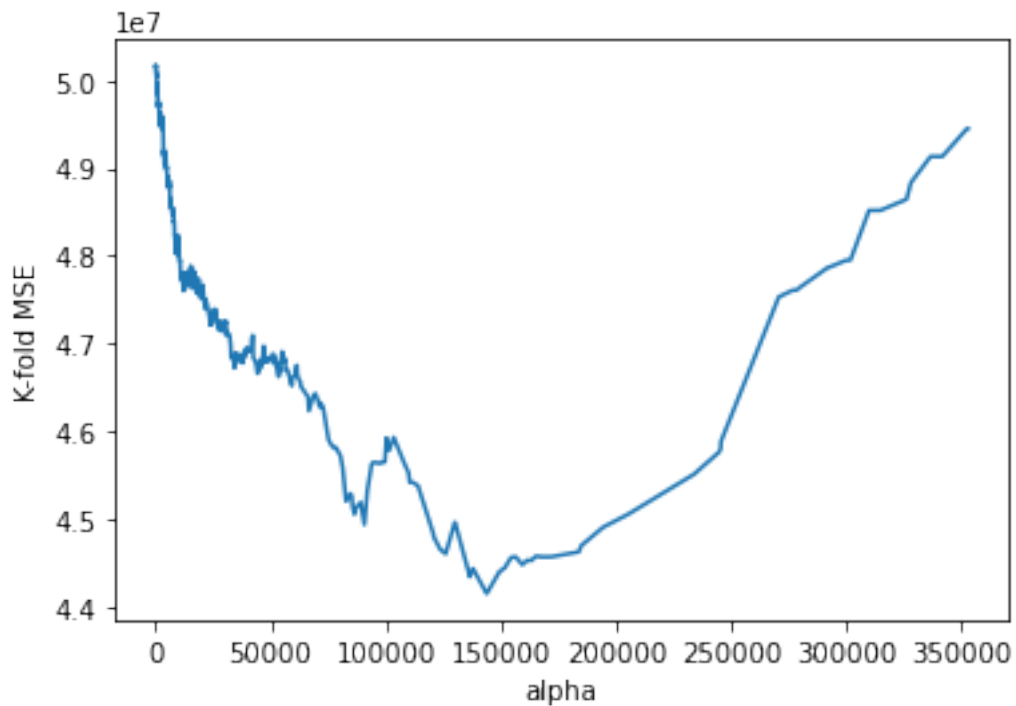
#Visualizing the 5-fold cross validation error vs alpha
plt.plot(alphas, cv_error)
plt.xscale('log')
plt.xlabel('alpha')
plt.ylabel('K-fold MSE');

```





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



### 3.3.1 Depth vs alpha; Node counts vs alpha

```

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

```

268.10325384140015

This code takes 4.5 minutes to run

```

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

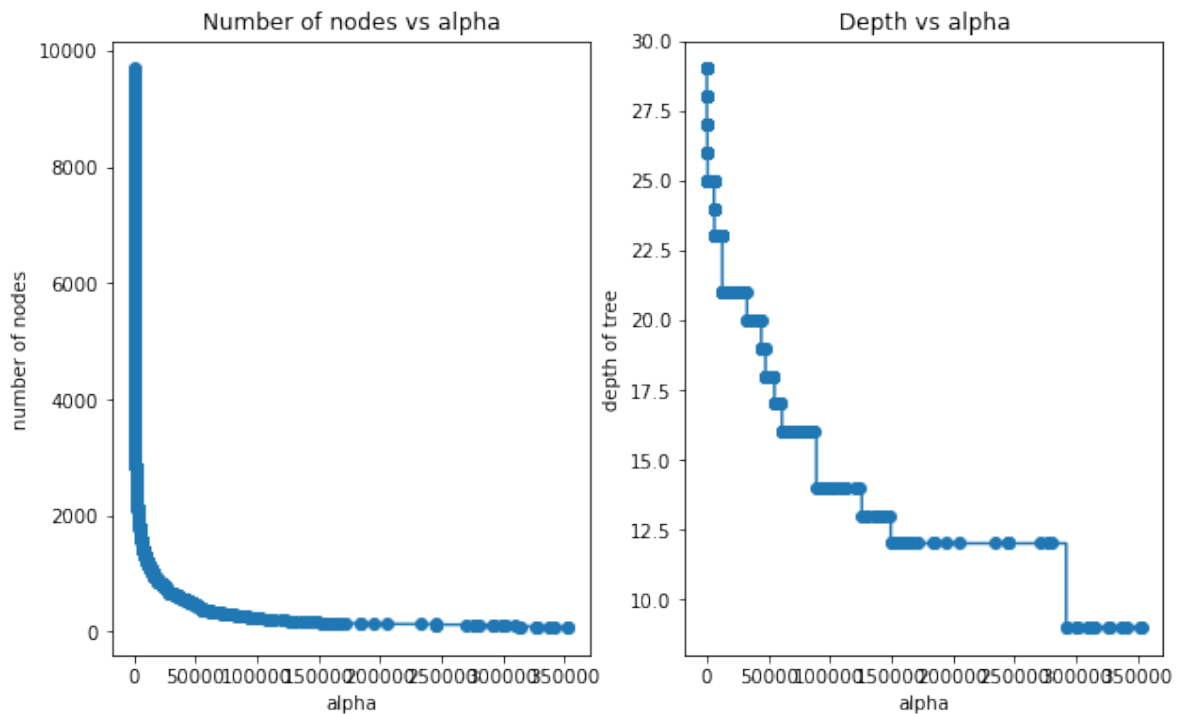
```

```

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

```

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



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

```

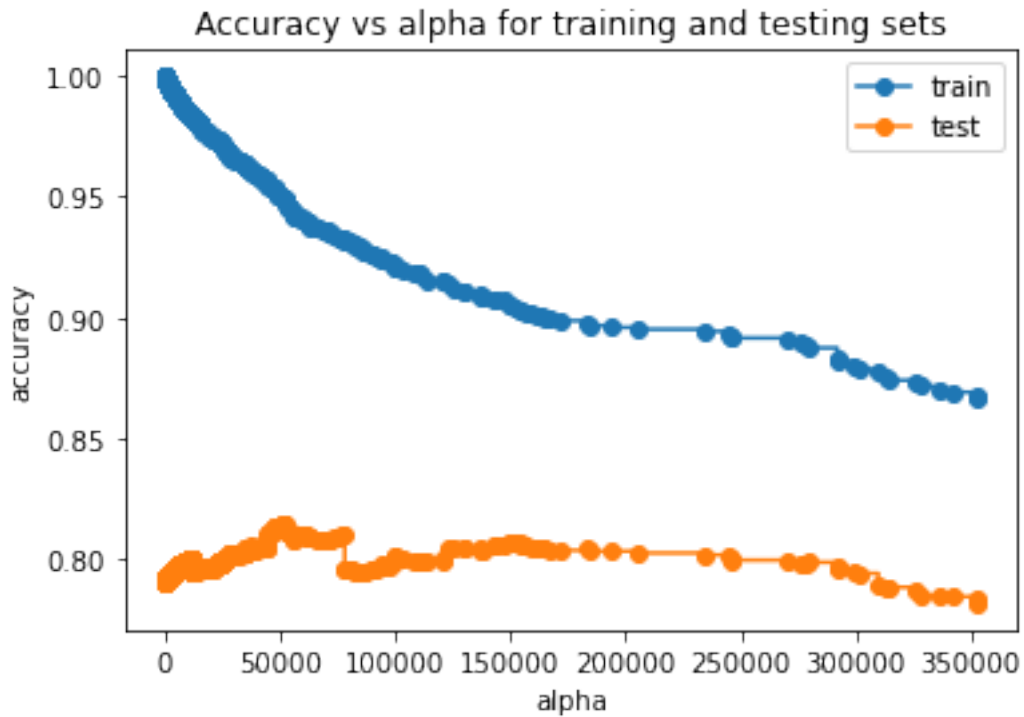
train_scores = [clf.score(X, y) for clf in trees]
test_scores = [clf.score(Xtest, test.price) for clf in trees]

```

```

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-post")
ax.plot(alphas[0:4093], test_scores[0:4093], marker="o", label="test", drawstyle="steps-post")
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_score
from sklearn.model_selection import StratifiedKFold, KFold
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import GridSearchCV

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

import time as time
```

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

```
test.head()
```

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

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

## 4.1 Building a classification tree

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

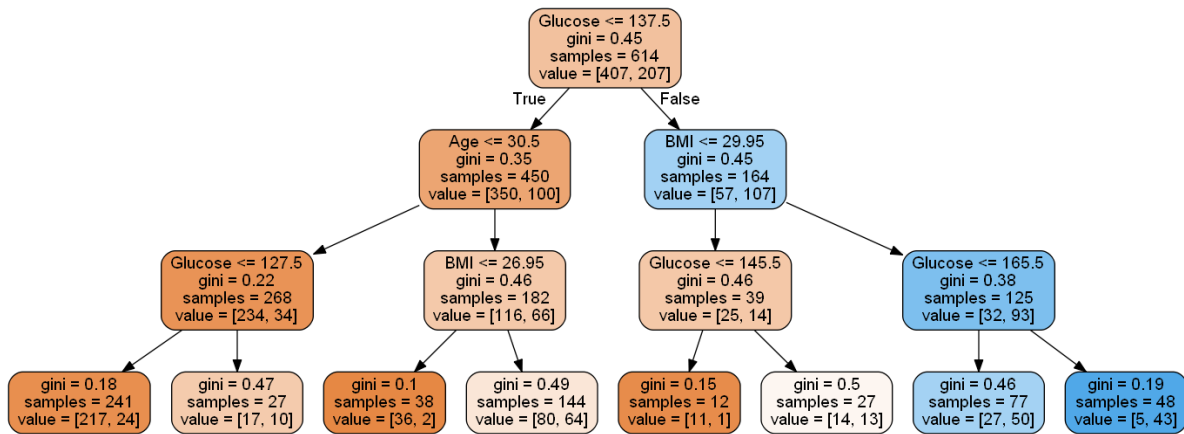
```
X = train.drop(columns = 'Outcome')
Xtest = test.drop(columns = 'Outcome')
y = train['Outcome']
ytest = test['Outcome']
```

```
#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'])
```

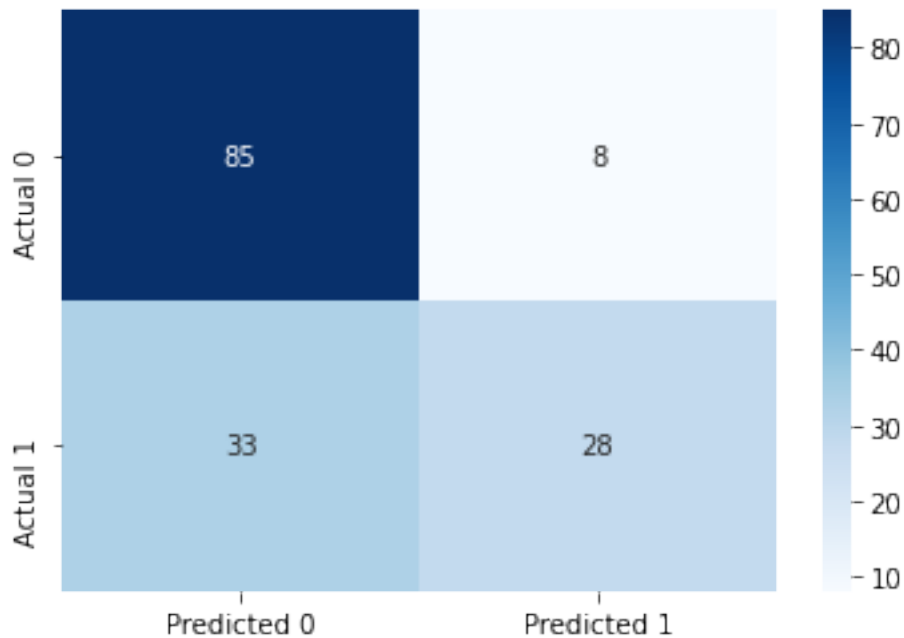
```
sns.heatmap(cm, annot=True, cmap='Blues', fmt='g');
```

```
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 nodes (leaves) that minimizes the FNR or maximizes recall.

Find the maximum values of depth and number of leaves.

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

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

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

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



118

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

14

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

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

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

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

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

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

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

```
# make the predictions
```

```
y_pred = grid_search.predict(Xtest)
```

```
print('Train accuracy : %.3f'%grid_search.best_estimator_.score(X, y))
```

```
print('Test accuracy : %.3f'%grid_search.best_estimator_.score(Xtest, ytest))
```

```
print('Best recall Through Grid Search : %.3f'%grid_search.best_score_)
```

```
print('Best params for recall')
```

```
print(grid_search.best_params_)
```

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

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

Train accuracy : 0.785

Test accuracy : 0.675

Best recall Through Grid Search : 0.658

Best params for recall

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

Time taken = 70 seconds

## 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.

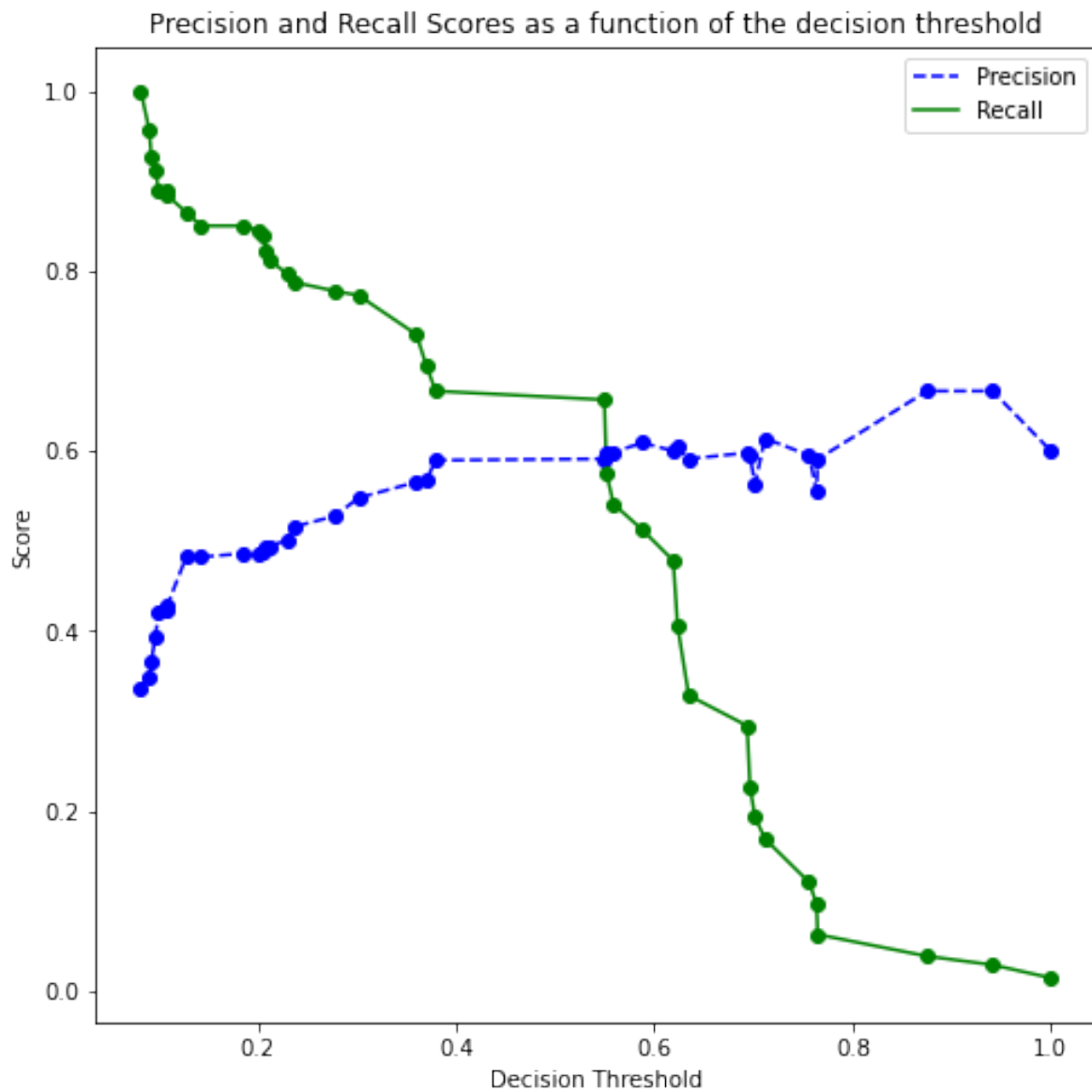
### 4.3.1 Balancing recall with precision

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

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

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

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



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

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

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

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

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

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

```

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

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

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

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

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

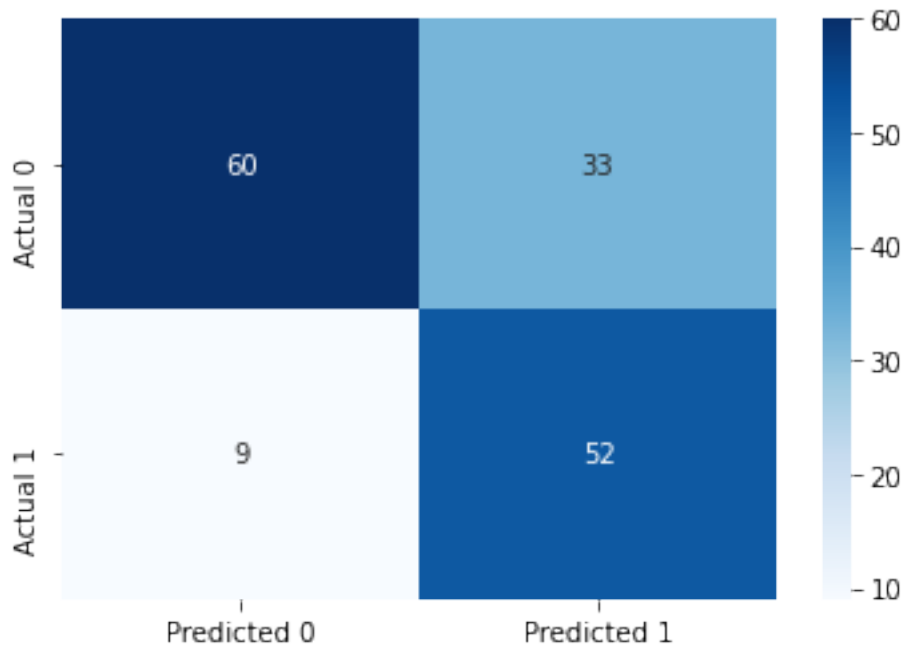
#Confusion matrix
cm = pd.DataFrame(confusion_matrix(ytest, y_pred),
                  columns=['Predicted 0', 'Predicted 1'], index = ['Actual 0', 'Actual 1'])
sns.heatmap(cm, annot=True, cmap='Blues', fmt='g');

```

```

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

```



#### 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 samples
```

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

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

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

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

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

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

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

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

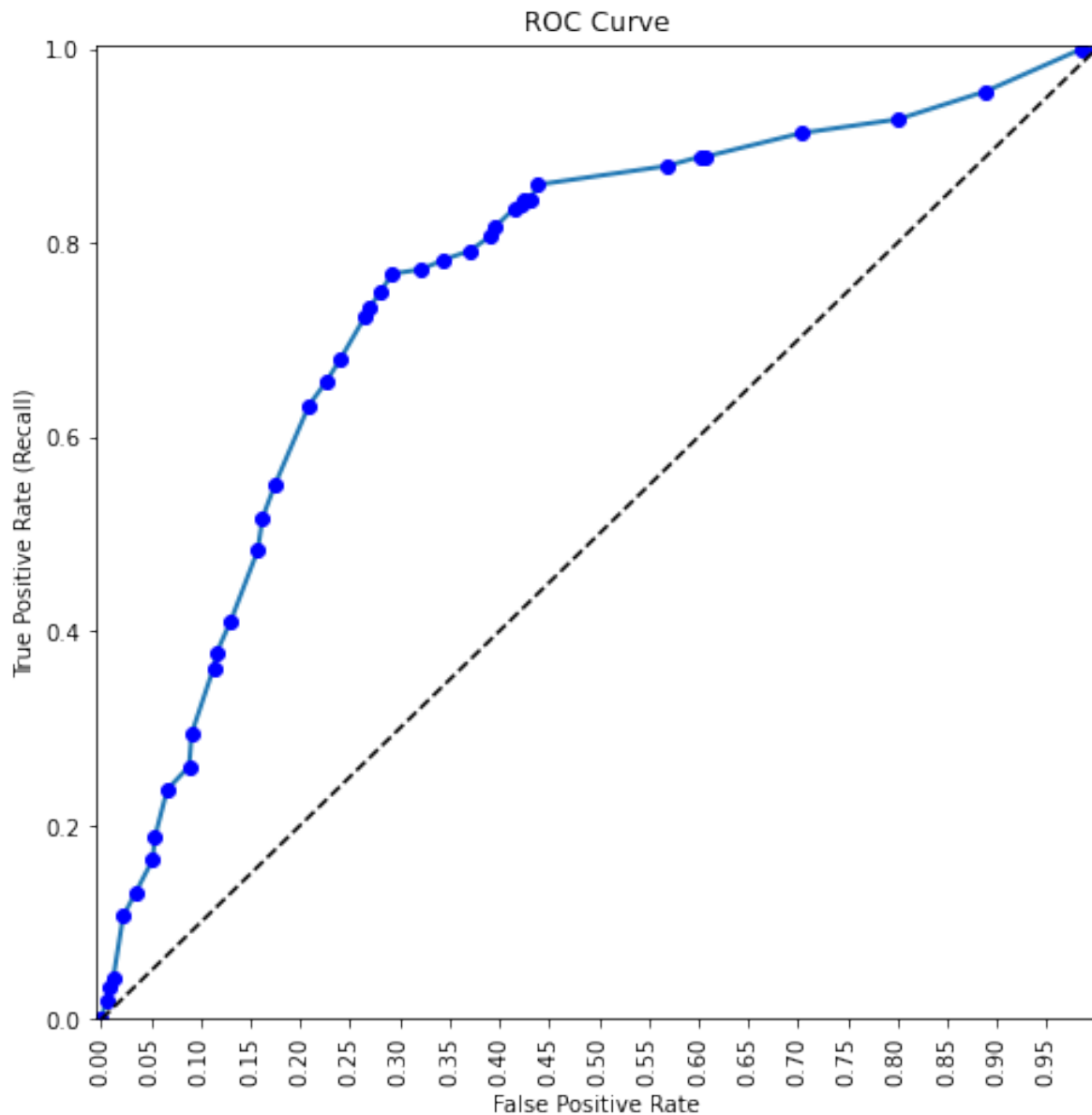
```
cross_val_ypred = cross_val_predict(DecisionTreeClassifier(random_state=1, max_depth = 6,
                                                           max_leaf_nodes=9, max_features=2)
                                   y, cv = 5, method = 'predict_proba')
```

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

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

0.7605075431162388





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

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

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

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

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

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

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

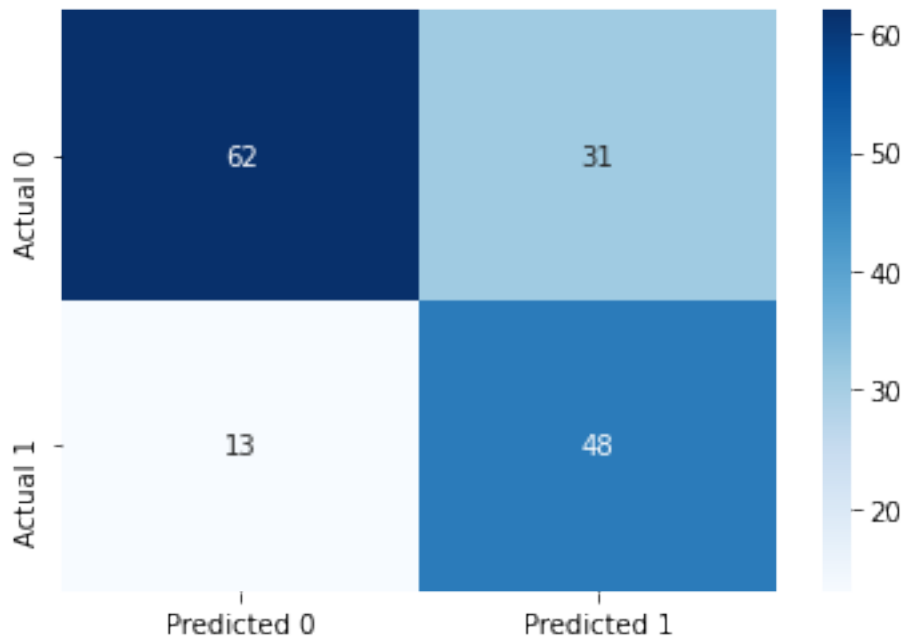
#Computing the accuracy
print("Accuracy: ",accuracy_score(y_pred, ytest)*100)

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

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

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

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



## 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 Cost-
```

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

58

```
#Grid search to optimize parameter values

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

grid_search.fit(X, y)
```

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

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

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

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

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

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

# A Stratified splitting (classification problem)

## A.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.

## A.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.

## A.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 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 test
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=data

#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  $x_1$ ,  $x_2$ , and  $y$ . This will tend to make the train and test datasets quite similar!

## A.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  $x_1$  and  $x_2$ , a continuous predictor  $x_3$ , 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_CatPredictors_stratified'})

# 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

# 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_state = i)
    model = LogisticRegression()
    model.fit(X_train, y_train)

    # Model accuracy on train and test data, with stratification only on response while splitting
    # 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 train and test data
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2, random_state = i,
                                                        stratify=pd.concat([x1, x2, y], axis=1))
    model.fit(X_train, y_train)

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

```
# 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()
```

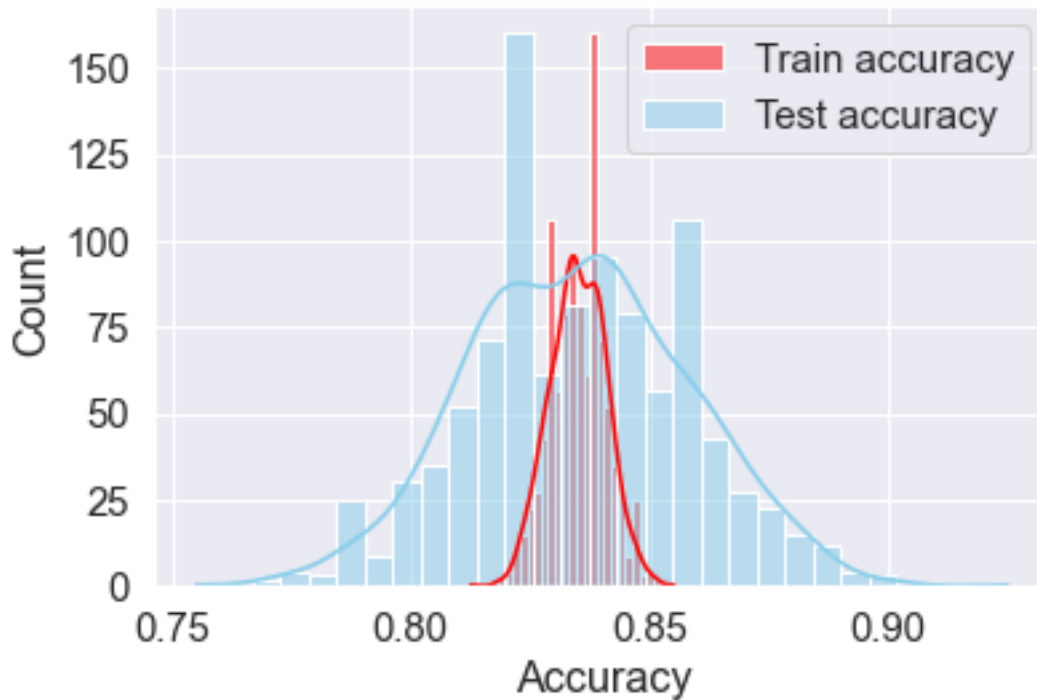
	train_y_stratified	test_y_stratified	train_y_CatPredictors_stratified	test_y_CatPredictors_stratified
count	999.000000	999.000000	9.990000e+02	9.990000e+02
mean	0.834962	0.835150	8.350000e-01	8.350000e-01
std	0.005833	0.023333	8.552999e-15	8.552999e-15
min	0.812500	0.755000	8.350000e-01	8.350000e-01
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.

### A.4.1 Stratified splitting only with respect to the response

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

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

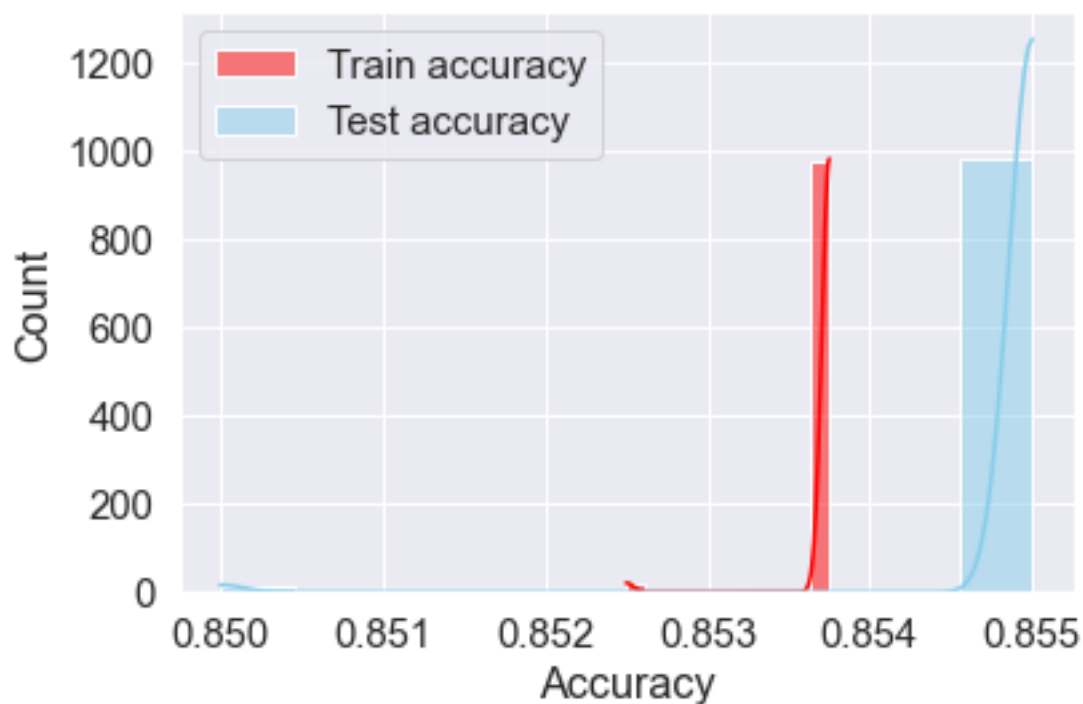


Note the variability in train and test accuracies when the data is stratified only with respect to the response. The train accuracy varies between 81.2% and 85.5%, while the test accuracy varies between 75.5% and 92.5%.

#### A.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="Train accuracy")
sns.histplot(data=accuracy_iter, x="test_y_CatPredictors_stratified", color="skyblue", label="Test accuracy")
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.

## **B Datasets, assignment and project files**

Datasets used in the book, assignment files, project files, and prediction problems report template can be found [here](#)