Task Document - Week 11

***Note to learner**: You MUST complete these exercises and upload your answers to the academy weekly, for your lecturer to review your progress and provide you feedback.*

Classification and Clustering Models

We begin this week's tasks using a variety of clustering and classification techniques. Please work through the following exercises, before attempting the tasks at the end of this document. Remember that the data used in the exercises has already been processed into a condition ready for use. When using these techniques on real-world data, it is essential that the full pre-processing stage is completed prior to any classification activities taking place.

Google Colab Notebook URL is below: (**Note**: the Colab title refers to Week 9 – please ignore this)

https://colab.research.google.com/drive/1X5Uf0Y8Zx7WuFnKi_jOwmkfwCkZtPS_a?usp=sharing

Learning Outcomes

Upon completion of this workbook, you will be able to:

- Be able to implement k-means clustering method to identify groupings within unlabelled datasets.
- Spilt a dataset into training and test data
- Create a Decision Tree model using Python
- Create a Random Forest model using Python
- Create a Support Vector Machine model using Python
- Understand how parameter tuning is able to make a model perform more effectively

Exercise 1 - K-Means Clustering

First, load up the "iris" dataset, then create a copy of this data without the "species" feature, called "cluster_data". [Note: the species feature stored separately as iris.target]

```
from sklearn import datasets
iris = datasets.load_iris()
cluster_data = iris.data
```

Once the data is in place, there a few packages that we need to proceed with the clustering and related tasks. Import following packages: **yellowbrick.cluster** and **sklearn.cluster**.

Before we begin with the clustering, we first need to establish a value for *k*, i.e. how many clusters is optimal for the data that we are working with. Today, we will use the "elbow" method to establish a value for *k*. To do that use the following command:

```
from yellowbrick.cluster import KElbowVisualizer
from sklearn.cluster import KMeans
model = KElbowVisualizer(KMeans(), k=10)
model.fit(iris.data)
```

The *KElbowVisualizer* function produces a chart use for determining the optimal number of clusters, which can be modified through a number of parameters. The first element of the function is an unfitted cluster i.e. KMeans or MiniBatchKMeans. The second term sets the parameter to be used for evaluating the optimal cluster number. The model is then fitted with values from dataset, or in straightforward terms, the data that is to be clustered.

When looking at the plot that is produced, you should be able to see that the point where the line begins to level off, creating the appearance of an "elbow" is at the point where the number of clusters is 3. We will therefore use 3 as our value for *k*.

To begin the process of the *k*-means clustering, we establish a data object called "clusters_k3", where the "k3" refers to the chosen value for *k*. To do this, we will use the *kmeans* function, setting the number of centroids at 3 (given here under the parameter "n_clusters"). Use the following command:

```
clusters_k3 = KMeans(n_clusters=3,init='k-
means++',n_init=10,max_iter=10,random_state=None)
y_kmeans = clusters_k3.fit_predict(iris.data)
```

Note that the *max_ iter* parameter constrains the number of iterations that are performed. When unbounded the algorithm will continue until convergence occurs. To check the locations of the three centroids, use the following command:

```
clusters_k3.cluster_centers_
```

This will provide the locations in the feature space where the 3 centroids are located after the 10 specified iterations have been completed. Now, to see the output of the clustering (i.e. which cluster each observation has been assigned to), use the following command:

```
clusters_k3.labels_
```

Observe the results of the clustering, and then compare it with the three classes that we know exist within the original iris data. You will most likely see that one cluster, representing the setosa plants, has been able to classify in a very accurate manner. However, it is likely that the other two classes may not have been clustered with the same degree of accuracy. Let's explore why that might be the case.

First ensure that **ggplot2** has been loaded into your R environment, and then try creating the following plots. The first will investigate the petal component:

```
from plotnine import *
import pandas as pd
import numpy as np

df = pd.DataFrame(data= np.c_[iris['data'], iris['target']], columns= iris['feature_names'] + ['target'])

ggplot(data = df) + geom_point(mapping = aes(x = 'petal length (cm)', y = 'petal width (cm)', color = 'target'))
```

When observing this plot, we can see that there are three distinct groups, so the issues is not with the petal dimensions. Now try the following command, to investigate the sepal component:

```
ggplot(data = df) + geom_point(mapping = aes(x = 'sepal length (cm)', y = 'sepal width (cm)', color = 'target'))
```

We can now see that the issue is due to the overlap within the natural groupings of two of the classes within the sepal dimensions. To further examine the issue, we can plot the centroid locations, using the feature coordinates of the two features pertaining to the sepal. Use the following commands:

```
cluster_centroids = pd.DataFrame (clusters_k3.cluster_centers_,columns= iris['feature_names'])

ggplot(data = None) + geom_point(data = df, mapping = aes(x = 'sepal length (cm)', y = 'sepal width (cm)', color = 'target')) + geom_point(data = cluster_centroids, mapping = aes(x = 'sepal length (cm)', y = 'sepal width (cm)'), color = "red", size = 4)
```

Notice that as we are creating a plot using two different data sets, we must set the data parameter at the *geom* level rather than at the global level. We can now see the location of the three cluster centroids and observe how each centroid attempt to find the optimum location for class membership identification.

We have now seen how clustering is able to assist in the labelling on unlabelled or unstructured data. Even when the results are not perfect, the value of clustering is that patterns and grouping within a dataset can be identified and used a basis for further investigation and analysis.

Exercise 2 - Data Pre-processing

One of the most essential phases of any machine learning task is ensuring that the data is in the optimal state for processing, prior to building any models. For this week's tasks, we will be using the **churn** dataset, which can be found in the online repository.

Download the file "churn.csv" from Blackboard and save it your working directory, and then load the data in your R environment as a data table called "churn":

```
import pandas as pd
churn = pd.read_csv('churn.csv')
```

We must ensure that there is no column with empty values. We will remove the rows having empty values under a given column using code lines below. Indexes of all rows having empty *TotalCharges* value is stored in a variable.

```
emptyRows = churn[churn['TotalCharges'].str.match(' ')].index

churn = churn.drop(emptyRows)
```

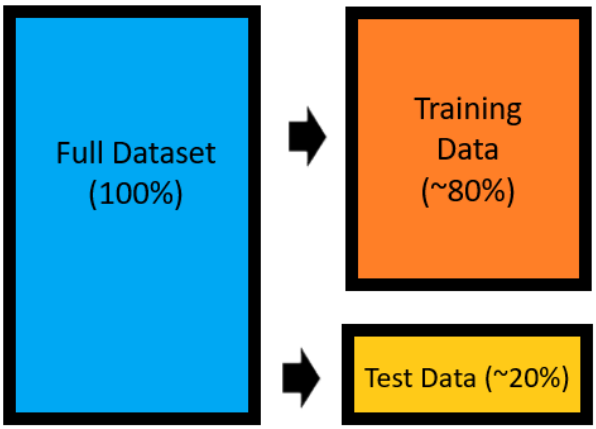
View the dataset, once you have loaded it in. You should notice that the data is primarily made up of data which is either Boolean or of a single digit order of magnitude. However, there are four features ("tenure", "Contract", "MonthlyCharges" and "TotalCharges") that have a significantly larger spread of values. To enable the machine learning algorithms to perform better, we will scale these features using the method used in previous weeks. Use the commands given below to write the appropriate function and apply that function to the "tenure" feature.

After this has been completed, apply the function to the other features that also require scaling.

```
from sklearn.preprocessing import MinMaxScaler

min_max_scaler = MinMaxScaler()

churn[['tenure']] = min_max_scaler.fit_transform(churn[['tenure']])

churn[['MonthlyCharges']] = min_max_scaler.fit_transform(churn[['MonthlyCharges']])

churn[['TotalCharges']] = min_max_scaler.fit_transform(churn[['TotalCharges']])
```

Machine learning models depend on at least two sets of data: training data and testing data (a third set – validation data – is sometimes also used to help ensure that the model will be useful when faced with unseen data). The training data provides the basis for a model to be trained, as in, the characteristics of data within this set are learned and associated with a particular class membership. The testing set provides data for the model to test, so that the efficacy and effectiveness of the model can be established.

Before we can begin to build any models, we must therefore split our dataset into two smaller subsets. The ratio of training to testing data will vary depending on a number of factors, including the size of the dataset itself (larger datasets allow for splits more heavily biased in favour of the training data, whereas smaller datasets may have to be more even to ensure that there are enough observations within the testing set), and the types of algorithms that is to be used. The most common splits are 70/30 and 80/20; for this exercise, we will use the latter.



To perform this split, a random number approach will be used. To achieve this within Python, we will allocate a pseudo-random value of either 1 or 0 to every observation within the dataset, with the probability of 0.8 for a 1, and 0.2 (or more accurately, 1 - 0.8), for 0. The *random_state* parameter of train_test_split function represent pseudo-random value. The second line divide the dataset up into the training and testing sets.

```
from sklearn.model_selection import train_test_split

trainset, testset  = train_test_split(churn, train_size=0.8, random_state=1)
```

Once the dataset has been divided up, the "customerID" feature should be removed, as the randomness inherent in that feature will inhibit the effectiveness of the classification. Use the following commands:

```
del trainset['customerID']

del testset['customerID']
```

Note that the ['customerID'] term refers to the removal of the customer ID feature in the dataset. This value should therefore be changed when dealing with datasets of differing dimensionality. Once the data has been split, we can then proceed to the next phase.

Exercise 3 - Decision Tree

Decision Trees are simple but robust classifiers that can be used with a variety of data types, that have the advantage of being easily interpretable. Furthermore, while we have performed some preparatory activities to ready the data for use, this is not always essential when using this method. Decision Trees are an intuitive concept; each "leaf" node represents a feature within the dataset, with each branch being the possible values (or ranges of values) associated with each feature. These can be mapped out – with class predictions attached – to create a tree-like structure.

To build a decision tree, we will use ***sklearn*** package for the model and to visualise the resulting tree. Import the package using following command:

```
from sklearn import tree
```

When building the decision tree, we need to encode feature values to numeric representation such as 'yes' and 'no' to 1 and 0. The *preprocessing* module of the *sklearn* library has function *LabelEncoder()* that can transform all string type feature values into numeric representation. The encoding of values is done using following command:

```
from sklearn import preprocessing

le = preprocessing.LabelEncoder()

trainsetEncoded = trainset.apply(le.fit_transform)

testsetEncoded = testset.apply(le.fit_transform)
```

The decision tree model can be initiated using *DecisionTreeClassifier()* function. The max_depth parameter is used to decide about depth of the tree.

```
DTModel = tree.DecisionTreeClassifier(max_depth = 3)
```

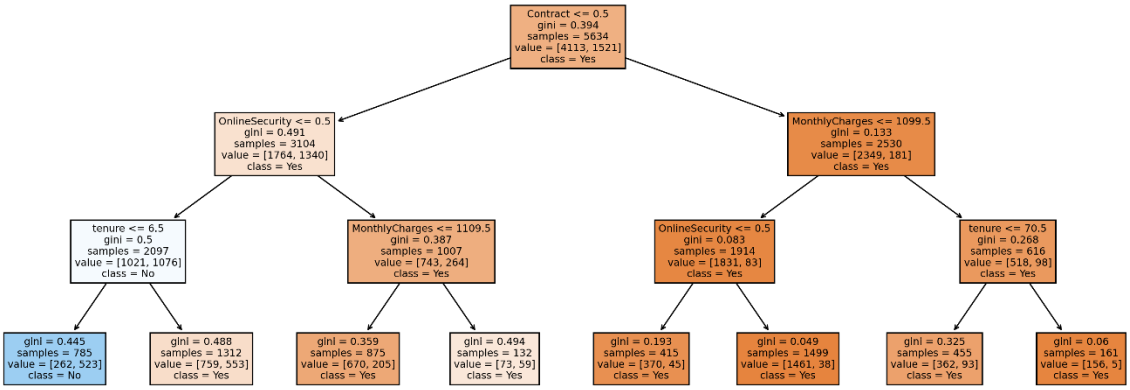
The decision tree model can be trained on testset using *fit()* function. The fit() function require features as first parameter and class labels or the target as second. Out churn trainset has 20

columns, where the last one is the class label. The following statements separates first 19 columns as features and the last column as target variable.

```
features = trainsetEncoded.iloc[: , :19]

target = trainsetEncoded.iloc[:,-1:]

DTModel = DTModel.fit(features, target )
```

Once the tree has been built, we can then visualise it to explore the results. Use the code given below.

```
from matplotlib.pyplot import figure
figure(figsize=(16, 6), dpi=80)
tree.plot_tree(DTModel)
plt.show()
```



The plot that is created should look similar to the one shown above. However, the feature name on which the decision tree has made a decision will not be shown. The right branch of every node represents true value while the left branch shows the false. Use the following code to display feature names along with their conditional values:

```
fn=list(features.columns.values)
cn=['Yes','No']
fig, axes = plt.subplots(nrows = 1,ncols = 1,figsize = (16,6), dpi=300)

tree.plot_tree(DTModel, feature_names = fn, class_names=cn, filled = True);
```

The trained decision tree can be interpreted as follows:

- Observations where the contract is longer will be classified as 0 (i.e., will not churn). This passes the "common sense check"; customers with plenty of time remaining on their contract are unlikely to leave.
- The customer subset that are most likely to churn are those with shorter contracts remaining and have low OnlineSecurity and tenure.

This ability to gather information from the tree that allows for decisions to be made more effectively is one of the key advantages of this approach. Once the model has been trained, the testing data must be prepared. For the model to be tested effectively, we must remove the response variable from the testing set, which can be performed with the following command:

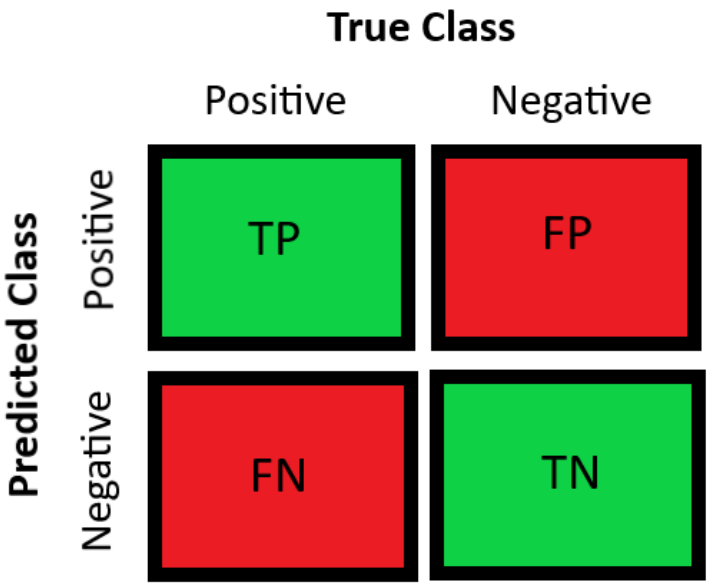
```
from sklearn import preprocessing

le = preprocessing.LabelEncoder()

testsetEncoded = testset.apply(le.fit_transform)
```

Next, we need to generate the predicted values, applying the testing data to the model. To achieve this, we can use the *predict* function of decision tree model. The *predict* function require features values of instances to predict.

```
testFeatures = testsetEncoded.iloc[: , :19]

testTarget = testsetEncoded.iloc[: , -1:]

predicted= DTModel.predict([testFeatures.iloc[3]])

print(predicted)
```

We are predicting the third row of the test set. You can compare the prediction accuracy by comparing *testTarget* of third row and final *predicted* output. Now that we have the predicted classes, we can create a confusion matrix, allowing use to assess the predictive performance of the model. A confusion matrix shows the values predicted by the model, compared with the actual label values of the training data. There are four possible combinations:

- If both values are 0 (no) then this is a true negative, or TN.
- If both values are 1 (yes) then this is a true positive, or TP.
- If the predicted value is 0, but the actual value is 1, then this is false negative, or FN.
- If the predicted value is 1, but the actual value is 0, then this is a false positive, or FP.



A confusion matrix shows the number of observations within each of the above groups. Use the *plot_ confusion_ matrix()* function to plot a confusion matrix:

```
from sklearn.metrics import plot_confusion_matrix

from sklearn.metrics import ConfusionMatrixDisplay

figure(figsize=(16, 6), dpi=80)

plot_confusion_matrix(DTModel, testFeatures,
testTarget,display_labels=cn,normalize='pred', cmap=plt.cm.Blues)

plt.title('Confusion Matrix')

plt.show()
```

This will populate a 2 x2 table, in the same format as the diagram shown above. Each window within the confusion matrix can be used to identify how well the classifier has performed, but as a metric for overall performance, we will use accuracy. The accuracy of a model is the total quantity of TP and TN results, over the total number of observations within the testing data. Use the following command to calculate the accuracy of the model:

```
from sklearn.metrics import accuracy_score
predictedAll= DTModel.predict(testFeatures)
accScore = accuracy_score(testTarget, predictedAll)
print(accScore)
```

This value can then be converted to a percentage, for example, a value of 0.79 can be read as 79%, meaning that the model was able to predict the correct class in 79% of cases.

Exercise 4 - Random Forest

While a Decision Tree is an effective and useful tool, it can be prone to overfitting at times and may not always deliver the best predictive performance. One method that can be used to improve upon these issues (at the cost of some computational efficiency) is the use of many Trees at once, a method known as a Random Forest. A Random Forest is an ensemble of Decision Trees, each with a subset of the total features used. The output from all the trees in the forest are either aggregated or averages to get a single output value for a particular data object.

```
from sklearn.ensemble import RandomForestClassifier

RFModel = RandomForestClassifier(n_estimators=500, max_features=5, max_depth=4,
random_state=0)

RFModel = RFModel.fit(features, target)
```

The first term in the *fit()* function identifies the training data. The second term specifies the target or the class labels of the training data. Within the initialization of random forest model, two additional parameters have been set: *max_ features*, which sets the number of features to be used per tree (note that this should never be set higher than the number of independent features that are present within the dataset) and *n_ estimators*, which specifies the number of trees that should be grown in the forest. So, in this example, there will be 500 trees within the forest, with each using 5 features.

Once the model has been trained, the *predict* function can again be used to apply to model to the testing data, so that the predictive capacity can be assessed. In this case, the type should be set to "response", as we are looking to produce a probabilistic class prediction.

```
RFpredicted= RFModel.predict(testFeatures)
```

Now we have the class predictions, we can produce another confusion matrix.

```
from sklearn.metrics import plot_confusion_matrix
from sklearn.metrics import ConfusionMatrixDisplay
figure(figsize=(16, 6), dpi=80)

plot_confusion_matrix(RFModel, testFeatures,
testTarget,display_labels=cn,normalize='pred', cmap=plt.cm.Blues)

plt.title('Confusion Matrix')

plt.show()
```

Again, we can use this as a basis for calculating the accuracy.

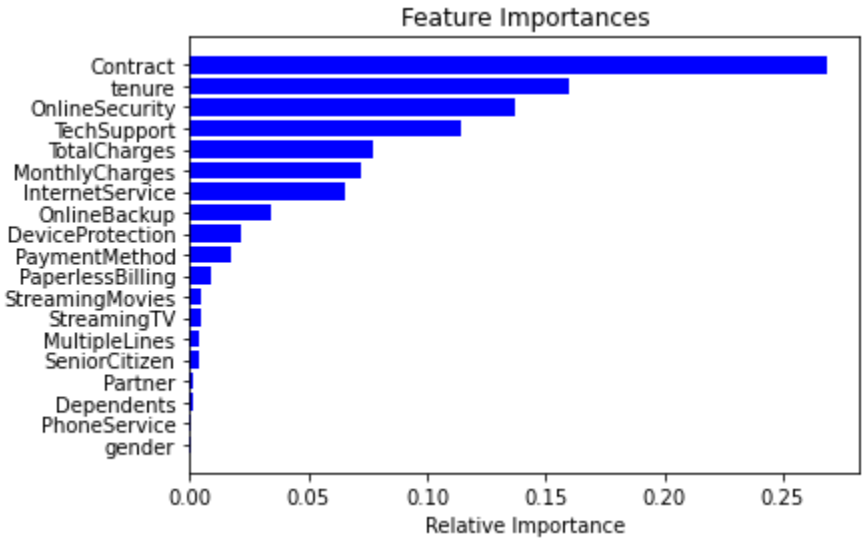
```
from sklearn.metrics import accuracy_score
accScore = accuracy_score(testTarget, RFpredicted)
```

This value can then be converted to a percentage, for example, a value of 0.801 can be read as 80.1%. This allows for a comparison to be made with other models that have been created so far.

Random Forests are useful because they also provide interpretability, as in, they can allow us to identify which features within the dataset are responsible for the greatest amount of variance. To do this, we can use the *feature_importances_* (i.e., variable importance), using the command below. We can then visualise the variable importance using the *barplot* command.

```
import numpy as np
importances = RFModel.feature_importances_
indices = np.argsort(importances)
plt.title('Feature Importances')


plt.barh(range(len(indices)), importances[indices], color='b', align='center')
plt.yticks(range(len(indices)), [fn[i] for i in indices])
plt.xlabel('Relative Importance')

plt.show()
```

This should produce a plot similar to the one shown below, where the tallest bars represent the features responsible for the most variance, or in other words, the features that contribute most to the predictions that are made by the model. Pleasingly, the three most important features (tenure, OnlineSecurity and contact) are the same as those identified by the decision tree, suggesting that both models are robust and are producing reliable results.



Exercise 5 - Support Vector Machine

To build a SVM model, first download and load in the package **svm** of the sklearn library. The creation of the training and testing data has already been done, so we can reuse the assets from earlier on in the process. The nature of an SVM as a true binary classifier, rather than a probabilistic classifier. The SVM model itself can be trained in a similar manner to the Random Forest. The second term in the function points toward the response variable, while the first directs the learner towards the data used for training.

```
from sklearn import svm

SvmModel = svm.SVC()

SvmModel = SvmModel.fit(features,target)
```

To explore the attributes of the model that has been identified, including the number of support vectors that have been generated, use the following command:

```
SvmModel.n_support_
```

We can use the *predict* function in the same way as it was used with the Random Forest model to generate a series of predicted values for the class of each data object. Use the following command:

```
SVMpredicted= SvmModel.predict(testFeatures)
```

The confusion matrix and the accuracy score can be produced in the same manner as before:

```
from sklearn.metrics import plot_confusion_matrix
from sklearn.metrics import ConfusionMatrixDisplay
figure(figsize=(16, 6), dpi=80)

plot_confusion_matrix(SvmModel, testFeatures,
testTarget,display_labels=cn,normalize='pred', cmap=plt.cm.Blues)

plt.title('Confusion Matrix')

plt.show()

from sklearn.metrics import accuracy_score

accScore = accuracy_score(testTarget, SVMpredicted)
```

Exercise 6 - Tasks

1. Try to build and test Naïve Bayes model yourself. To construct a Naïve Bayes model, we will once again use the **GaussianNB** module of the sklearn. naive_bayes library. This should be imported into your environment.

   The *process* is straightforward, with the key command being *fit()*, as shown below:

   ```
   gnbModel = GaussianNB()
   gnbModel = gnbModel.fit(X,y)
   ```

   Where *y* is the target variable and *x* is the input data. What predictive accuracy did you obtain through this approach?

2. The process of tuning the parameters of a model to attempt to maximise the performance is called parameter optimisation. Complete the table below, tuning the parameters for the random forest with values of your choosing and recording the results (the first row represents the example completed in the exercise). Which combination performs the best?

Attempt Number	*max_ features*	*n_ estimators*	Accuracy (%)
1	5	500	
2			
3			
4			

3. One of the key parameters that can be tuned when creating an SVM model is the choice of kernel that is used. There are three forms of kernel that can be used: rbf, linear, poly and sigmoid. For example, to use a linear kernel, the following command would be used:

   ```
   SvmModel = svm.SVC(kernel='rbf',gamma='scale')
   ```

   Complete the table below, along with the example from the exercise. Which kernel performs the best?

Attempt Number	Kernel	Accuracy (%)
1	rbf kernel	
2	Linear kernel	
3	Polynomial kernel	
4	Sigmoid kernel	

4. Knowing the characteristics of the machine learning methods that have been used, and the characteristics of the data, why do you think that method that produced the best results was able to outperform the other, in terms of predictive capacity?

Regression Models

We continue this week's tasks with a variety of regression-based learning methods. Please work through the following exercises, before attempting the tasks at the end of this document. Remember that the data used in the exercises today has already been processed into a condition ready for use. When using these techniques on real-world data, it is essential that the full pre-processing stage is completed prior to any modelling being performed.

Google Colab Notebook URL is below:

https://colab.research.google.com/drive/1hFjbJhieIqKcvRwnKW5kco-E2FJ6oORZ?usp=sharing

Exercise 7 - Linear Regression

The most fundamental form of regression model is univariate linear regression, where a model is built using a single independent variable which has a linear relationship with the dependent. In Python, ***sklearn*** machine learning library has the ***linear_ model*** module that contains linear regression function.

First, load in the "iris" dataset. Once that has been loaded, use only Petal length to predict Petal Width. The **LinearRegression()** construct a basic linear model of the relationship between the features "petal length" and "petal width". We first slice the "petal length" and "petal width" columns into *irisX* and *irisY* variables. Use the following command:

```
from sklearn import datasets, linear_model
iris = datasets.load_iris()


irisX = iris.data[:, 3] #3 represent petal length
irisX = irisX.reshape(-1,1)
irisY = iris.data[:, 2] #2 represent petal width
irisY = irisY.reshape(-1,1)
```

```
lrModel = linear_model.LinearRegression()
lrModel.fit(irisX, irisY)
```

Once the model has been created and trained, we can look at the coefficient values that have been generated. Use the **lrMode.coef_** and **lrMode.intercept_** to print this information to the console:

```
print(lrModel.coef_)
print(lrModel.intercept_)
```

You should see the values for the intercept and the coefficient for "petal width", i.e. the *x* value. If you were to transpose these values into the equation for the line of best fit, you should get:

$$y = 2.2x + 1.08$$

This equation therefore describes the relationship between the two features, in linear terms. Use the linear regression model to predict the "petal length" by providing the "petal width" as input.

```
predictY = lrModel.predict(irisX)
```

To verify that the model has produced the correct findings, first plot the data onto a scatter plot, and then map the regression line onto the plot. Use the following commands:

```
plt.scatter(irisX, irisY,  color='black')
plt.plot(irisX, predictY, color='blue', linewidth=2)
plt.title('Linear Regression - Iris dataset')
plt.xlabel('Petal Width (cm) - X')
plt.ylabel('Petal Length (cm) - y')

plt.show()
```

You should see that the steepness of the slope and the value of the intercept correspond with the output of the model. Before we can make any predictions using the model, however, we must check the model fit, to ensure that the predictive capacity of the model is sufficient to make reliable predictions. To do this, use the **statmodels** library to print the details of the model to the console. Use the following command:

```
import statsmodels.api as sm

X2 = sm.add_constant(irisX)
est = sm.OLS(irisY, X2)
```

```
est2 = est.fit()
print(est2.summary())
```

The two statistics that we are looking for are the adjusted r-squared value, and the p-value. For the r-squared, which describes model fit, remember that a value nearer to 1 denotes a good fit, while a value nearer to 0 denotes a poor fit. For this model, the adjusted r-squared value should be around 0.927, indicating that the model does indeed fit the data well.

The feature significance, given by the p-value, allows us to identify whether the relationship found between the independent and dependent features within the sample used to build the model is generalisable to the population. A p-value of less than 0.05 indicates a statistically significant feature, and the p-value for this model should be well below that threshold, giving us confidence when using this feature in the model.

Now we will apply this technique to a larger dataset. Load in the "diamonds" data, and then split the dataset into a training and test dataset using the same method as in last week's workshop. The code below shows the commands to use, modified to fit the structure and format of the "diamonds" data:

```
import pandas as pd
from sklearn.model_selection import train_test_split

url = 'https://github.com/tidyverse/ggplot2/raw/master/data-raw/diamonds.csv'

diamonds = pd.read_csv(url)
trainset , testset   = train_test_split(diamonds, train_size=0.8, random_state=1)
```

Once we have the separate training and testing datasets, we can use the training data to build a linear model to model the relationship between the feature's "price" and "carat", with price as the dependent variable.

```
lrModel = linear_model.LinearRegression()

carat = trainset[['carat']]
price = trainset[['price']]

lrModel.fit(carat,price)
```

Use the **statsmodels** to explore the model in more detail. What values did you obtain for the r-squared and the p-value?

Now that we are satisfied that the model has sufficient predictive capacity, we can try and predict the values using the testing data. To do this, we can use the **predict** function. Use the following command:

```
price_pred = lrModel.predict(price)
```

As we are not using a class-based output, we cannot test the results of the model using a method that depends on a confusion matrix, such as accuracy or the F1 score. Instead, we can use the Pearson correlation coefficient to identify the alignment between the actual prices of the diamonds within the testing dataset, and the prices predicted by the model, by using the **corr** function. Use the following command:

```
import pandas as pd

df = pd.DataFrame(data=price_pred,columns=['predPrice'])

df2 = df.merge(testset[['price']],left_index=True, right_index=True)

corr = df2.corr(method='pearson')

corr
```

Remember that a correlation coefficient closer to 1 indicates a closer fit. What value did you obtain? What does that tell you about the model?

Exercise 8 - Multiple Regression

Multivariate regression, or multiple regression, uses the same concept of linear regression, but allows for more than one independent feature to be utilised. In many cases, this allows for a model to be built that has a greater predictive capacity, and one which is able to model the real-world more closely.

In Python, we still use the **linear_model** module of ***sklearn*** library to construct a multiple regression model; the only difference is the number of features which are added into the command. For example, to build a model that is able to predict the price of a diamond using the features "carat", "x", "y" and "z", use the following command:

```
lrModel = linear_model.LinearRegression()

features = trainset[['carat','x','y','z']]
price = trainset[['price']]

lrModel.fit(features,price)
```

As before, use the **statsmodel** library to explore the coefficients and the model fit. What do you notice about the r-squared value, now that more features have been added to the model?

Use the same **predict** function to predict the values again, and assess the results using the **corr** function, as seen in the commands below.

```
price_pred = lrModel.predict(price)
```

Has the performance been improved now that the additional features have been added? What does the fact that the change is quite small tell you about the information gain offered by the features that have been used?

Random Forest Regression

Random Forest Regression is a flexible and robust form of regression model, that can be used in wide variety of circumstances, with a wide variety of data and data types.

To do this, we need to import the packages named **RandomForestRegressor** from **sklearn.ensemble**. Once these packages have been imported, we can proceed with building the model. To perform this, use the following command:

```
from sklearn.ensemble import RandomForestRegressor
RFreg = RandomForestRegressor()
RFreg.fit(features,price)
```

Next, we must predict the testset using trained regression model using the following command:

```
testFeatures = testset[['carat','x','y','z']]
price_pred = RFreg.predict(testFeatures)
```

We can see from the adjusted r-squared value by using r2_score function. The evaluation score of the Random Forest Regressor shows an improvement as compared to linear regression model.

```
from sklearn.metrics import r2_score
print("%.2f" % r2_score(testset[['price']], price_pred))
```

There are a several parameters that can be used to tune performance of the Random Forest Regressor. The purpose of some parameters are as follows:

- **n_estimators**: The number of trees in the forest.
- **min_samples_split**: The minimum number of samples required to split an internal node. This parameter is sensitive to data type of the features. For int type data, it consider the minimum number of the feature values whereas, for the float type data instances, it consider ceil() of the values.
- **min_samples_leaf**: The minimum number of samples required to be at a leaf node. A split point at any depth will only be considered if it leaves at least min_samples_leaf training samples in each of the left and right branches. This may have the effect of smoothing the model, especially in regression.
- **max_depth**: The maximum depth of the tree. If None, then nodes are expanded until all leaves are pure or until all leaves contain less than min_samples_split samples.

In addition, you can perform exhaustive search over specified parameter values for an estimator. This will provide the estimated best or the most suitable parameters for the Random Forest Regressor. To get which parameters are best for your dataset, use following code:

```
from sklearn.model_selection import RandomizedSearchCV

n_estimators = [int(x) for x in np.linspace(10,200,10)]
max_depth = [int(x) for x in np.linspace(10,100,10)]
min_samples_split = [2,3,4,5,10]
min_samples_leaf = [1,2,4,10,15,20]
random_grid = {'n_estimators':n_estimators,'max_depth':max_depth,
        'min_samples_split':min_samples_split,'min_samples_leaf':min_samples_leaf}
```

```
RFrandom = RandomizedSearchCV(estimator=RFreg, param_distributions=random_grid,cv =
3)


RFrandom.fit(features,price)

y_pred = RFrandom.predict(testFeatures)

print(RFrandom. best_params_)
```

Exercise 9 - Tasks

For this task, ensure that **sklearn** has been imported and as we will be using a dataset that comes pre-loaded into this package. In today's tasks, we will be using some of the skills we have developed so far, including data exploration, feature engineering, feature selection and model tuning. Firstly, however, load in the "Boston" dataset. Once that has been loaded in, there are two preliminary tasks:

i. Keep the target feature of the Boston dataset as the response variable. The target column of the Boston dataset contains the median value of owner-occupied homes in $1000's.

ii. Sum any two features (columns) of the dataset to form a new feature and use it for further tasks.

Once this has been performed, you should then continue onto the following tasks.

1. Using the technique used previously, split the data into training and test sets. Remember to update the code to reflect the name of the new dataset (i.e. boston)
2. Select one of the continuous features and construct a linear model, using "target" as the response variable. After building a univariate model, add build a multivariate model by adding another independent feature, from the continuous features available within the dataset. Continue this pattern one more time and populate the table below. Note that IF1 through IF3 denote the independent features that have been chosen.

Model Attempt	IF1 name	Is significant?	IF2 name	Is significant?	IF3 name?	Is significant?	R-squared	Model accuracy (as correlation coefficient)
1								
2								
3								

3. Build a Random Forest Regressor using this data. Record the parameters that were used for the RF regressor.

    3.1. Check the importance of features using Randomized Search, and their standardised beta coefficients. What does this tell you about the features?

    3.2. Check the predictive performance of the model. How does it compare with the linear models?

If you have time, feel free to go back to the previous Random Forest Regressor used for the "diamonds" data, and experiment with the various parameters used when setting up the model, to assess what impact changing their values has on the model performance.

There are several other regression models in the sklearn library such as Logistics Regression, Bayesian Regression, Stochastic Gradient Descent – SGDRegression, Ridge regression, Lasso and more. They all have the fit(X,y) function. You can try them and evaluate their performance.