

# Table of Contents

<b>Chapter 1: 8. Performance evaluation methods</b>	1
<b>Why is performance evaluation critical?</b>	1
<b>Performance evaluation for classification algorithms</b>	2
Confusion Matrix	3
Normalized Confusion Matrix	6
Area under the curve	8
Cumulative Gains Curve	10
Lift Curve	13
KS Statistic plot	16
Calibration Plot	18
Learning Curve	20
Cross-validated box plot	21
<b>Performance evaluation for regression algorithms</b>	23
Mean Absolute Error (MAE)	23
Mean Squared Error (MSE)	24
Root Mean Squared Error (RMSE)	25
<b>Performance evaluation for unsupervised algorithms</b>	25
Elbow plot	26
<b>Summary</b>	27
<b>Index</b>	29

---

# 1

## 8. Performance evaluation methods

Performance evaluation varies by the type of machine learning algorithm you choose to implement. In general, you have different metrics that can potentially determine how well your model is performing at it's given task for classification, regression and unsupervised machine learning algorithms.

In this chapter you will explore how the different performance evaluation methods help you understand your model better. The chapter is split into three sections:

1. Performance evaluation for classification algorithms
2. Performance evaluation for regression algorithms
3. Performance evaluation for unsupervised algorithms

### Why is performance evaluation critical?

The key aspect here is to understand why we need to evaluate the performance of a model in the first place. Some of the potential reasons as to why performance evaluation is critical are as follows:

1. **Prevents Overfitting:** Overfitting occurs when your algorithm hugs the data too tightly and therefore makes predictions that are specific to that dataset only. In other words, your model cannot generalize it's predictions well outside the data it was trained on.
2. **Prevents Underfitting:** The exact opposite of overfitting. In this case, the model is very generic in nature.
3. **Understanding Predictions:** Performance evaluation methods will help you understand, in greater detail as to how your model makes predictions & the nature of these predictions along with useful information such as the accuracy of

your model.

## Performance evaluation for classification algorithms

In order to evaluate the performance of for classification let's consider the two classification algorithms that we have already built during the course of this book - the K-Nearest Neighbors & the logistic Regression.

The first step is to implement both of these algorithms on the fraud detection dataset. We can do this by using the code shown below:

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsClassifier
from sklearn import linear_model

#Reading in the fraud detection dataset

df = pd.read_csv('fraud_prediction.csv')

#Creating the features

features = df.drop('isFraud', axis = 1).values
target = df['isFraud'].values

#Splitting the data into training and test sets

X_train, X_test, y_train, y_test = train_test_split(features, target,
test_size = 0.3, random_state = 42, stratify = target)

# Building the K-NN Classifier

knn_classifier = KNeighborsClassifier(n_neighbors=3)

knn_classifier.fit(X_train, y_train)

#Initializing an logistic regression object

logistic_regression = linear_model.LogisticRegression()

#Fitting the model to the training and test sets

logistic_regression.fit(X_train, y_train)
```

In the code above, we have read the fraud detection dataset into our notebook & have split the data into the features and target variables as usual. We then split the data into training and test sets and build the K-Nearest Neighbors and Logistic Regression models on the training data.

In this section you will learn how to evaluate the performance of a single model - The K-Nearest Neighbors. You will also learn how to compare and contrast multiple models. Therefore, you will learn about the:

1. Confusion Matrix
2. Normalized Confusion Matrix
3. Area Under the Curve (AUC Score)
4. Cumulative Gains Curve
5. Lift Curve
6. KS Statistic Plot
7. Calibration Plot
8. Learning Curve
9. Cross-Validated Box Plot

Some of the visualizations in this section require a package titled - "scikit-plot". The scikit-plot is a very effective package that is used to visualize the various performance measures of machine learning models and is specifically built for the models that are built using scikit-learn.

In order to install the scikit-plot on your local machine using pip in your terminal we use the code shown below:

```
pip3 install scikit-plot
```

If you are using the Anaconda distribution in order to manage your python packages you can install scikit-plot by using the code shown below:

```
conda install scikit-plot
```

## Confusion Matrix

Up until now we used the accuracy as the sole measure of model performance. This was fine because we have a balanced dataset. A balanced dataset is one in which we have almost the same number of labels for each category. In the dataset that we are working with 8000 labels belong to the fraudulent transactions while 12,000 belong to the non-fraudulent transactions.

Imagine a situation in which 90% of our data had non-fraudulent transactions while only 10% of the transactions had fraudulent cases. If the classifier reported an accuracy of 90% it wouldn't make sense because most of the data that it has seen thus far were the non-fraudulent cases and it has seen very little of the fraudulent cases. So even if it classifies 90% of the cases accurately it means that most of the cases that it classifies would belong to the non-fraudulent cases. This provides no value to us.

A confusion matrix is a performance evaluation technique that can be used in such cases where we do not have a balanced dataset. The confusion matrix for our dataset would look like the image illustrated below:

	<b>PREDICTED: FRAUD</b>	<b>PREDICTED: NOT FRAUD</b>
<b>ACTUAL: FRAUD</b>	<b>TRUE POSITIVE</b>	<b>FALSE NEGATIVE</b>
<b>ACTUAL: NOT FRAUD</b>	<b>FALSE POSITIVE</b>	<b>TRUE NEGATIVE</b>

Confusion Matrix for fraudulent transactions

The goal of the confusion matrix is to maximize the number of True Positives and True Negatives as this gives the correct predictions and to minimize the number of False Negatives and False Positives as this gives us the wrong predictions.

Depending on your problem the false positives might be more problematic than the false negatives and vice versa and thus the goal of building the right classifier should be to solve your problem in the best possible way.

In order to implement the confusion matrix in scikit-learn we use the code shown below:

```
from sklearn.metrics import confusion_matrix

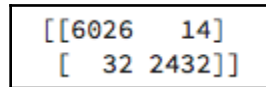
#Creating predictions on the test set
```

```
prediction = knn_classifier.predict(X_test)

#Creating the confusion matrix

print(confusion_matrix(y_test, prediction))
```

This produces an output as illustrated below:



```
[[6026  14]
 [ 32 2432]]
```

The confusion matrix output of our classifier for fraudulent transactions

In the code above we create a set of predictions use the `.predict()` method on the test training data and then we used the `confusion_matrix()` function on the test set of the target variable and the predictions created earlier.

The confusion matrix above looks almost perfect as most cases are classified into the True positive and True negative along the main diagonal. Only 46 cases are classified wrongly and this number is almost equal. This means that the number of False Positives and False Negatives are minimal and balanced and one does not outweigh the other. This is an example of the ideal classifier.

Three other metrics that can be derived from the confusion matrix are **precision**, **recall** and **F1-Score**. A high value of precision indicates that not many non-fraudulent transactions are classified as fraudulent, while a high value of recall indicates that most of the fraudulent cases were predicted correctly.

The F1-Score is weighted average of the precision and recall.

We can compute the precision and recall by using the code shown below:

```
from sklearn.metrics import classification_report

#Creating the classification report

print(classification_report(y_test, prediction))
```

This produces an output as illustrated in the image below:

	precision	recall	f1-score	support
0.0	0.99	1.00	1.00	6040
1.0	0.99	0.99	0.99	2464
avg / total	0.99	0.99	0.99	8504

Classification Report

In the code above we use the `classification_report()` function with two arguments the test set of the target variable and the prediction variable that we created earlier for the confusion matrix.

In the output the precision, recall & F1-score are all high because we have built the ideal machine learning model. These values range from 0 to 1, with 1 being the highest.

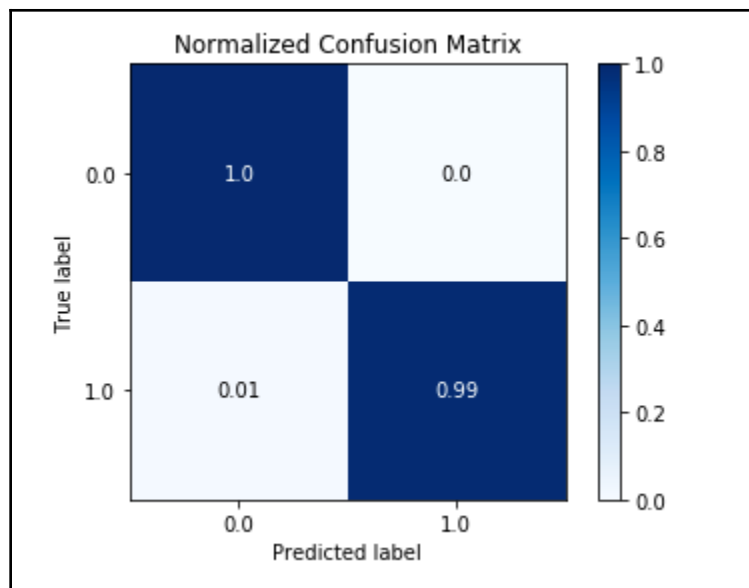
## Normalized Confusion Matrix

A normalized confusion matrix makes it easier for the data scientist to visually interpret how the labels are being predicted. In order to construct a normalized confusion matrix we use the code shown below:

```
#Normalized confusion matrix for the K-NN model

prediction_labels = knn_classifier.predict(X_test)
skplt.metrics.plot_confusion_matrix(y_test, prediction_labels,
normalize=True)
plt.show()
```

This results in a confusion matrix as illustrated in the image below:



Normalized confusion matrix for the K-NN model

In the plot above, the predicted labels are along the x-axis while the true or actual labels are along the y-axis. We see that the model has got 0.01 or 1% of the predictions for the fraudulent transactions wrong while 0.99 or 99% of the fraudulent transactions have been predicted correctly. We can also see that the K-NN model predicted 100% of the non fraudulent correctly.

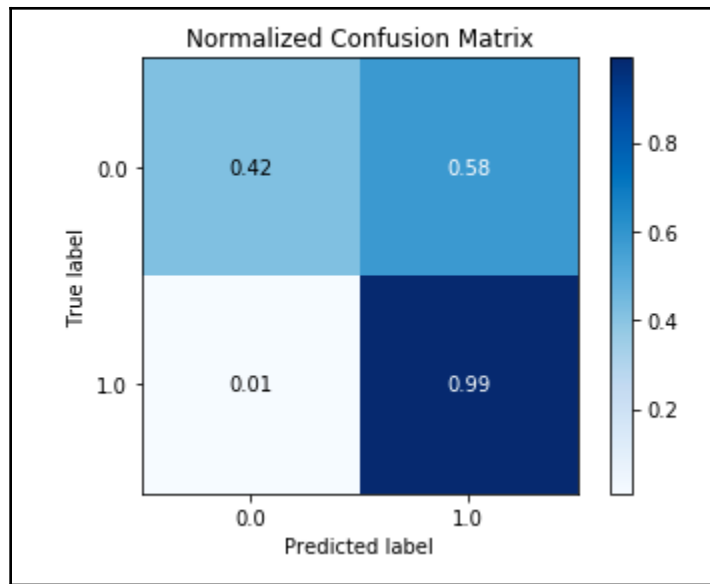
We can now compare the performance of the logistic regression model using a normalized confusion matrix by using the code shown below:

```
#Normalized confusion matrix for the logistic regression model

prediction_labels = logistic_regression.predict(X_test)
skplt.metrics.plot_confusion_matrix(y_test, prediction_labels,
normalizer=True)
plt.show()
```

This results in a confusion matrix as illustrated in the image below:





Normalized confusion matrix for the logistic regression model

In the confusion matrix above it is clear that the logistic regression model only predicted 42% of the non fraudulent transactions correctly. This indicates, almost instantly the the K-Nearest Neighbor model performed better.

## Area under the curve

The 'curve' in this case is the ROC curve or the Receiver Operator Characteristics curve. This is a plot between the True Positive Rate and the False positive Rate. We can plot this curve by using the code shown below:

```
from sklearn.metrics import roc_curve
from sklearn.metrics import roc_auc_score
import matplotlib.pyplot as plt

#Probabilities for each prediction output

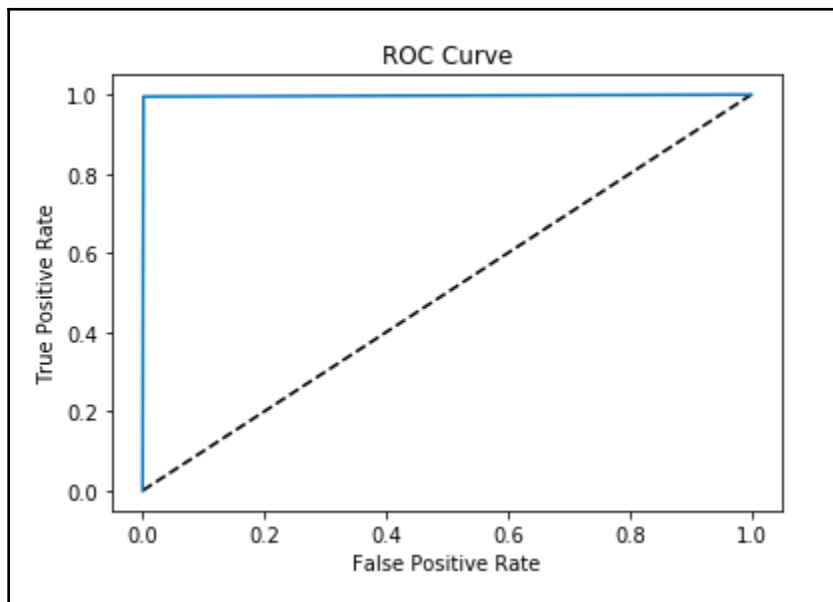
target_prob = knn_classifier.predict_proba(X_test)[:,-1]

#Plotting the ROC curve

fpr, tpr, thresholds = roc_curve(y_test, target_prob)
```

```
plt.plot([0,1], [0,1], 'k--')  
  
plt.plot(fpr, tpr)  
  
plt.xlabel('False Positive Rate')  
  
plt.ylabel('True Positive Rate')  
  
plt.title('ROC Curve')  
  
plt.show()
```

This produces a curve as illustrated below:



ROC curve

In the code above we first create a set of probabilities for each of the predicted labels. For instance, the predicted label of '1' would have a certain set of probabilities associated with them while the label '0' would have a certain set of probabilities associated with them. Using these probabilities we use the `roc_curve()` function along with the target test set to generate the ROC curve.

The curve you see above is an example of the perfect ROC curve that you will see. The curve above has a True Positive Rate of 1.0 which indicates accurate predictions while having a False Positive Rate of 0 which indicates the lack of wrong predictions.

Such a curve also has the most area under the curve compared to curves of models that have a lower accuracy. In order to compute the Area Under the Curve score we use the code shown below:

```
#Computing the auc score  
  
roc_auc_score(y_test, target_prob)
```

This produces a score of 0.99. A higher auc score is indicate of a better performing model.

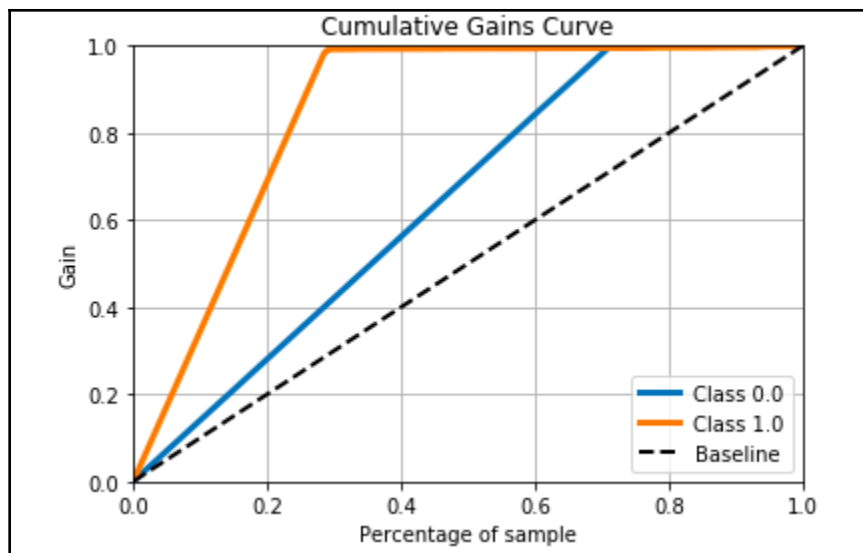
## Cumulative Gains Curve

When building multiple machine learning models, it is important to understand which of the models in question produces the type of predictions that you want it to generate. The cumulative gains curve helps you with the process of model comparison by telling you about the percentage of a category/class that appears within a percentage of the sample population for a particular model.

In simple terms, in the fraud detection dataset - we might want to pick a model that can predict a larger number of fraudulent transactions opposed to model that cannot. In order to construct the cumulative gains plot for the K-Nearest Neighbors model we use the code shown below:

```
import scikitplot as skplt  
  
target_prob = knn_classifier.predict_proba(X_test)  
skplt.metrics.plot_cumulative_gain(y_test, target_prob)  
plt.show()
```

This results in a plot as illustrated in the image below:

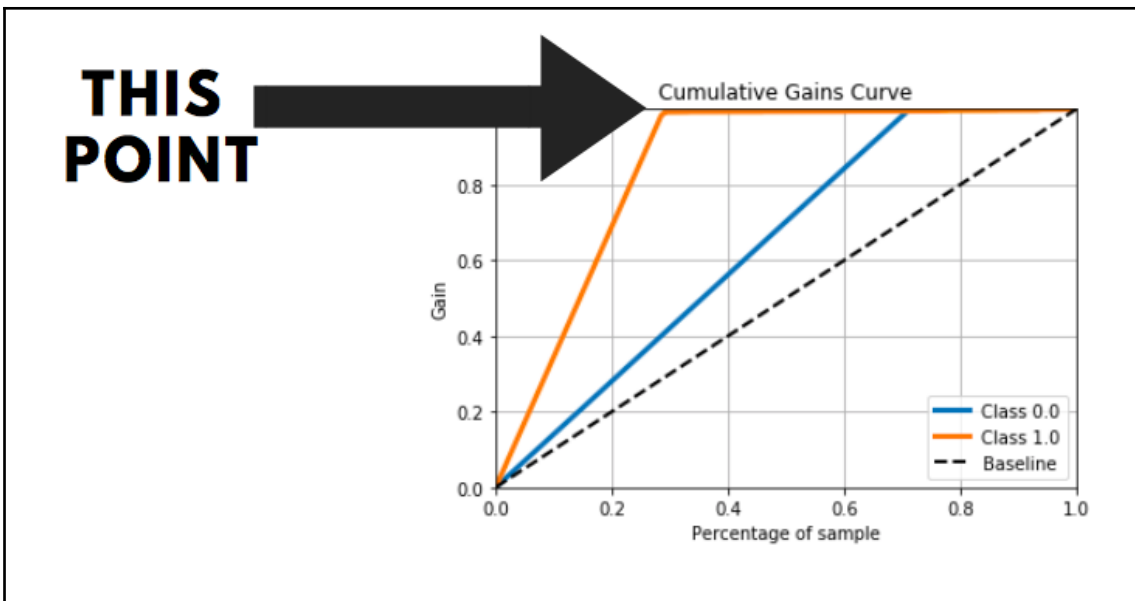


Cumulative gains plot for the K-Nearest Neighbors model

In the code above:

- We first import the scikit-plot package which generates the plot illustrated above. We then compute the probabilities for the target variable which in this case is the probabilities if a particular mobile transaction is fraudulent or not on the test data.
- Finally, we use the `plot_cumulative_gain()` function on these probabilities & the test data target labels in order to generate the plot that is illustrated above.

How do we interpret the plot above? We simply look for the point at which a certain percentage of the data contains 100% of the target class. This point is illustrated in the image below:



Point at which 100% of the target class exists

The point described in the image above corresponds to a value of 0.2 on the x-axis and 1.0 on the y-axis. This means that 0.2 to 1.0 or 20% to 100% of the data will consist of the target class 1 which is the fraudulent transactions.

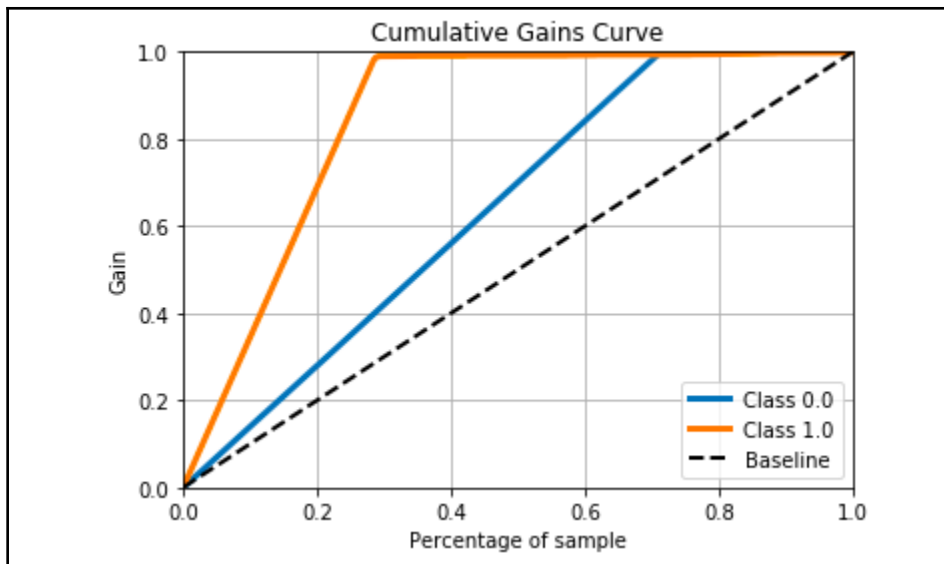
This can also be interpreted as - 80% of the total data will contain 100% of the fraudulent transaction predictions if you use the K-Nearest Neighbors model.

Let's now compute the cumulative gains curve for the logistic regression model and see if it is different. In order to do this we use the code shown below:

```
#Cumulative gains plot for the logistic regression model

target_prob = logistic_regression.predict_proba(X_test)
skplt.metrics.plot_cumulative_gain(y_test, target_prob)
plt.show()
```

This results in a plot as illustrated in the image below:



Cumulative gains plot for the Logistic Regression Model

The plot above is similar to the cumulative gains plot that was previously produced by the K-NN model in that 80% of data contains 100% of the target class. Therefore, using either the K-NN or the logistic regression model will yield similar results.

It is however a good practice to compare how different models behave using the cumulative gains chart in order to gain a fundamental understanding of how your model makes predictions.

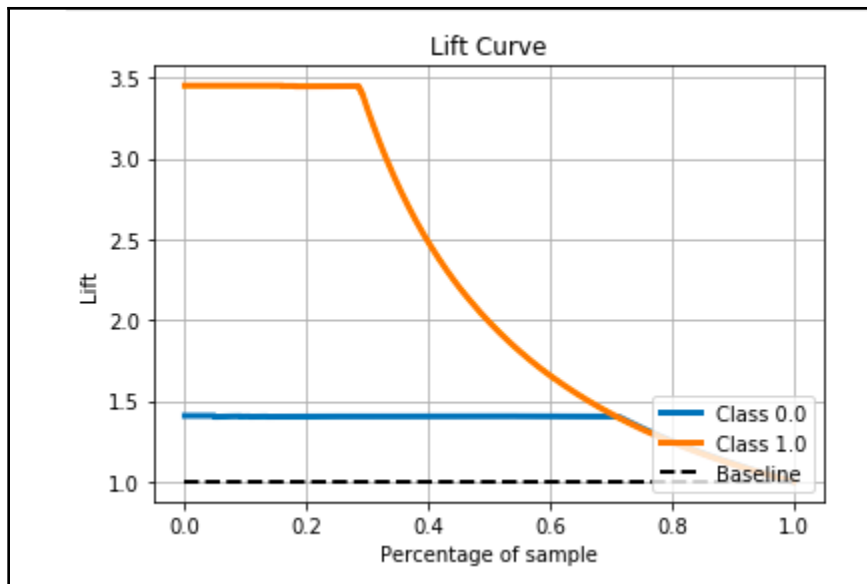
## Lift Curve

A lift curve fundamentally gives you information about how well you can make predictions by using a machine learning model opposed to when you are not using one. In order to construct a lift curve for the K-Nearest Neighbor model, we use the code shown below:

```
# Lift curve for the K-NN model

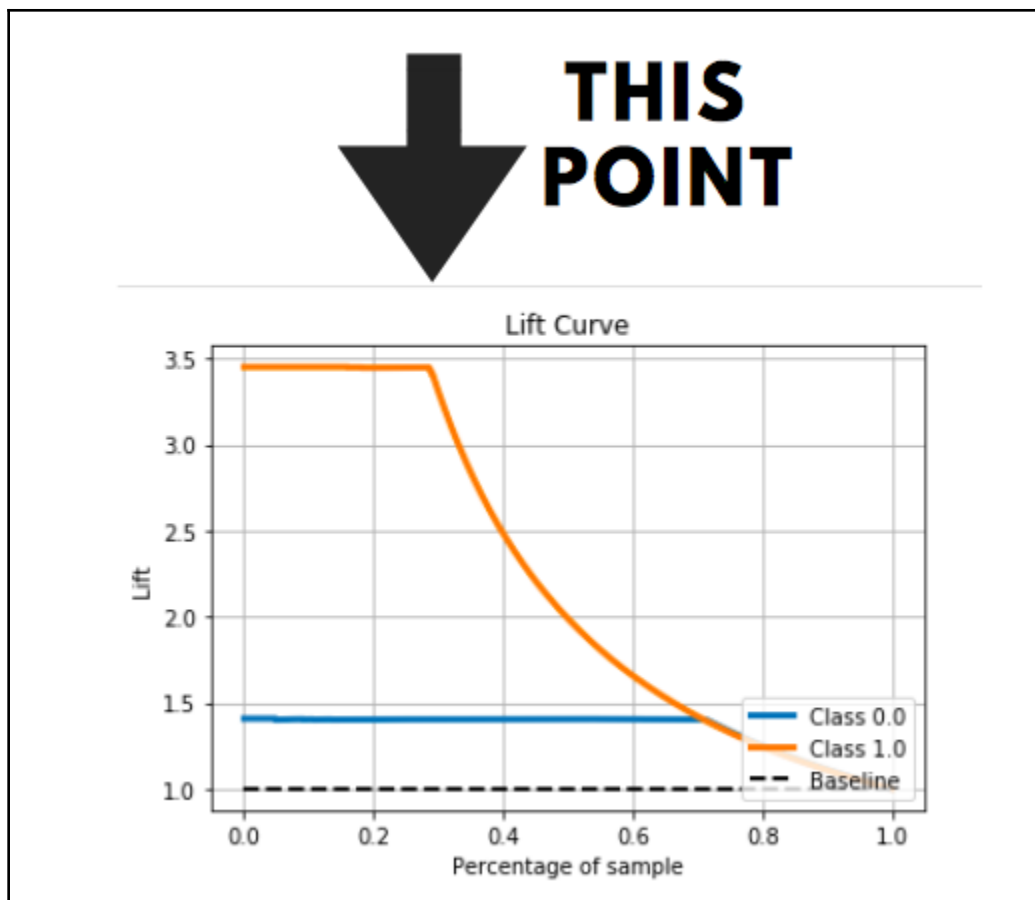
target_prob = knn_classifier.predict_proba(X_test)
skplt.metrics.plot_lift_curve(y_test, target_prob)
plt.show()
```

This results in a plot as illustrated in the image below:



Lift curve for the K-NN model

How do we interpret the lift curve illustrated above? We need to look for the point in the curve at which the curve dips. This is illustrated for you in the figure shown below:



Point of interest in the lift curve

In the plot above, the point highlighted is the point that we want to look for in any lift curve. The point tells us that 0.3 or 30% of our total data will perform 3.5 times better when using the K-NN predictive model opposed when we do not use any model at all in order to predict the fraudulent transactions.

We can now construct the lift curve for the logistic regression model in order to compare and contrast the performance of the two models. We can do this by using the code shown below:

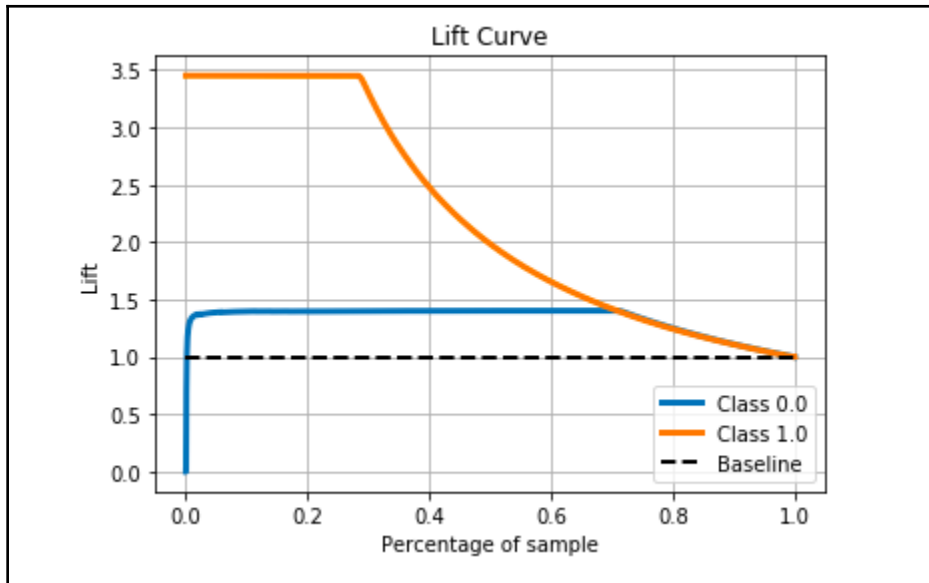
```
#Cumulative gains plot for the logistic regression model

target_prob = logistic_regression.predict_proba(X_test)
skplt.metrics.plot_lift_curve(y_test, target_prob)
```



```
plt.show()
```

This results in a plot as illustrated in the image shown below:



Lift curve for the logistic regression model

Although the plot tells us that 30% of the data will see an improved performance similar to that of the K-NN model that we built earlier in order to predict the fraudulent transactions there is a difference when it comes to predicting the non fraudulent transactions (the blue line).

For a small percent of the data the lift curve for the non-fraudulent transactions is actually lower than the baseline (the dotted line). This means that the logistic regression model does worse than not using a predictive model for a small percentage of the data when it comes to predicting the non-fraudulent transactions.

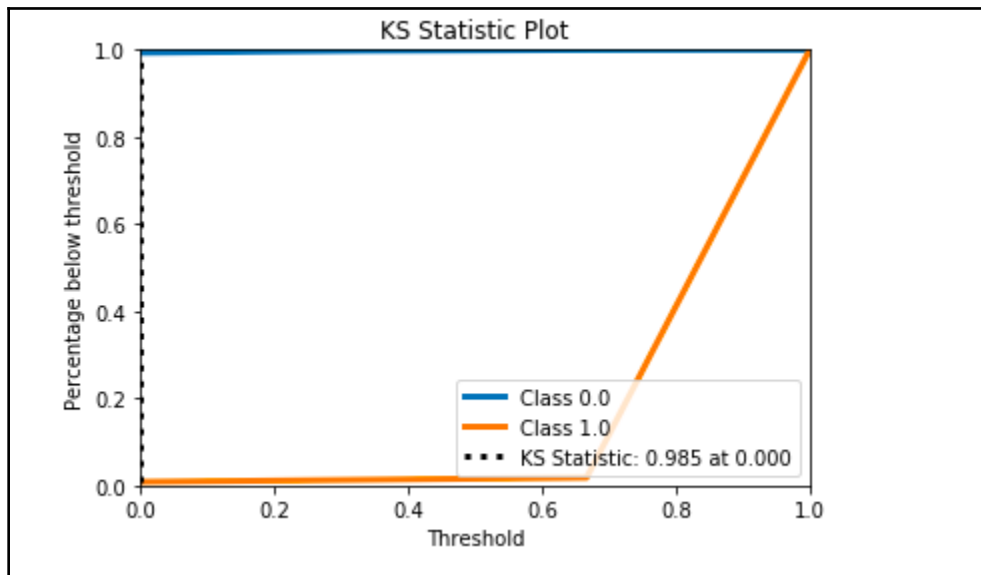
## KS Statistic plot

The K-S Statistic plot or the Kolomogorov Smirnov statistic plot is a plot that tells you if the model gets confused or not when it comes to predicting the different labels in your dataset. In order to understand what the 'term' confused means in this case we will construct the K-S statistic plot for the K-NN model by using the code shown below:

```
#KS plot for the K-NN model

target_proba = knn_classifier.predict_proba(X_test)
skplt.metrics.plot_ks_statistic(y_test, target_proba)
plt.show()
```

This results in a plot as illustrated in the image below:



K-S Statistic plot for the K-NN model

In the plot above:

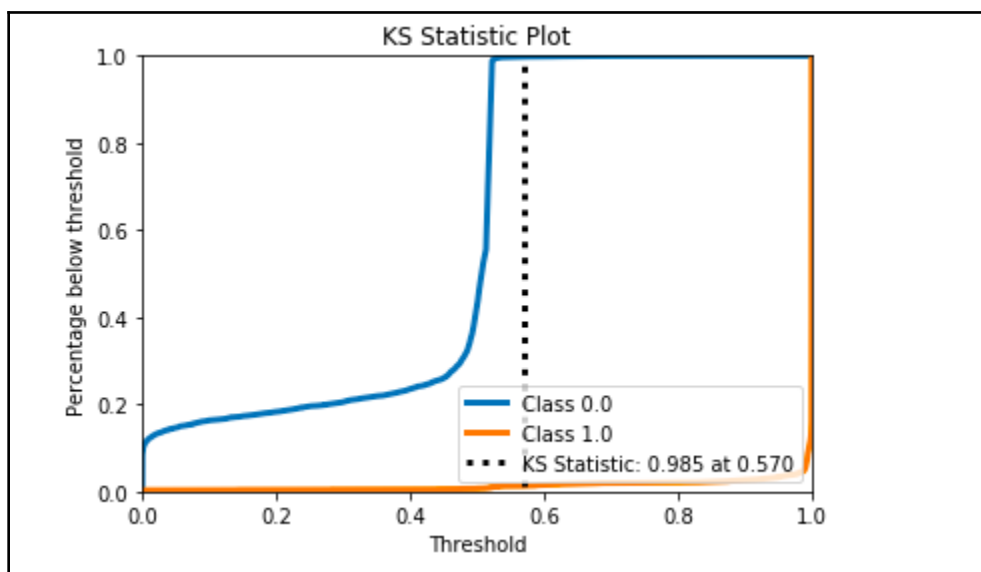
1. the dotted line is the distance between the predictions for the fraudulent transactions (Yellow line on the bottom) and the non fraudulent transactions (blue line on top). This distance is 0.985 as indicated by the plot.
2. A KS statistic score that is close to 1 is usually a good indication that the model does not get confused between predicting the two different target labels and can make a clear distinction when it comes to predicting the labels.
3. In the plot above, the score of 0.985 can be observed to be difference between the two classes of predictions for upto 70% (0.7) of the data. This can be observed along the x-axis as a threshold of 0.7 still has the maximum separation distance.

We can now compute the K-S statistic plot for the logistic regression model in order to compare which of the two models provides a better distinction in predictions between the two class labels. We can do this by using the code shown below:

```
#KS plot for the logistic regression model

target_proba = logistic_regression.predict_proba(X_test)
skplt.metrics.plot_ks_statistic(y_test, target_proba)
plt.show()
```

This results in a plot as illustrated in the image shown below:



KS Statistic plot for the logistic regression model

Although the two models have the same separation score of 0.985, the threshold at which the separation occurs is quite different. In the case of the logistic regression, this distance only occurs for the bottom 43% of the data since the maximum separation starts at a threshold of 0.57 along the x-axis.

This means that the K-Nearest Neighbors model which has a large distance for about 70% of the total data is much better at making predictions about fraudulent transactions.

## Calibration Plot

A calibration plot, as the name suggests tells you how well your model is 'calibrated'. The well calibrated model will have a prediction score equal to the fraction of positive class (in this case, the fraudulent transactions). In order to plot a calibration plot we use the code shown below:

```

#Extracting the probabilities that the positive class will be predicted

knn_proba = knn_classifier.predict_proba(X_test)
log_proba = logistic_regression.predict_proba(X_test)

#Storing probabilities in a list

probas = [knn_proba, log_proba]

# Storing the model names in a list

model_names = ["k_nn", "Logistic Regression"]

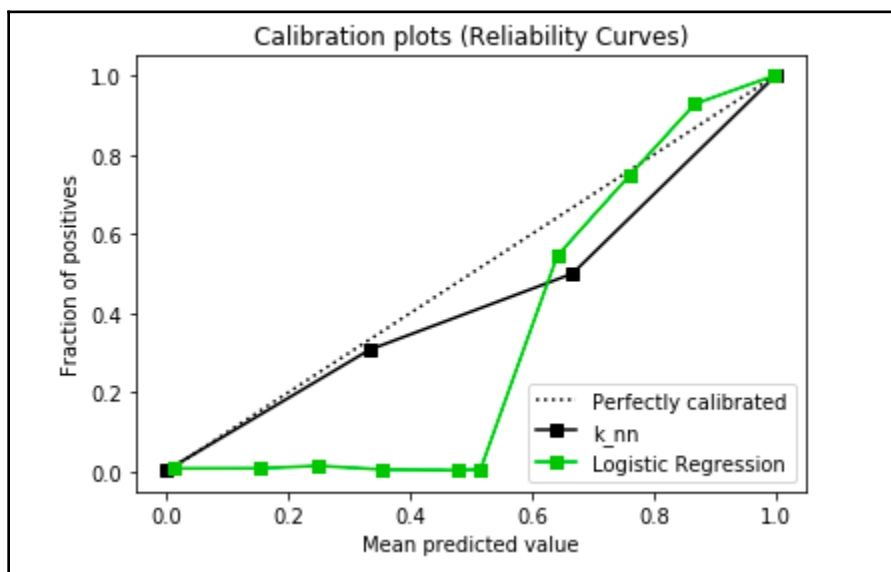
#Creating the calibration plot

skplt.metrics.plot_calibration_curve(y_test, probas, model_names)

plt.show()

```

This results in a calibration plot as illustrated in the image below:



Calibration plot for the two models

In the code above:

1. We first compute the probability that the positive class (fraudulent transactions) will be predicted for each model.

2. We then store these probabilities and the model names in a list.
3. Finally, we use the `plot_calibration_curve()` function from the `scikit-plot` package with these probabilities, the test labels and the model names in order to create the calibration plot.

This results in the calibration plot as illustrated above:

1. The dotted line represents the perfect calibration plot. This is because the mean prediction value has the exact value as the fraction of the positive class at each and every point.
2. From the plot it is clear that the K-Nearest Neighbors model is much better calibrated than the calibration plot of the logistic regression model.
3. This is because the calibration plot of the K-Nearest model follows that of the ideal calibration plot much more closely than the calibration plot of the logistic regression model.

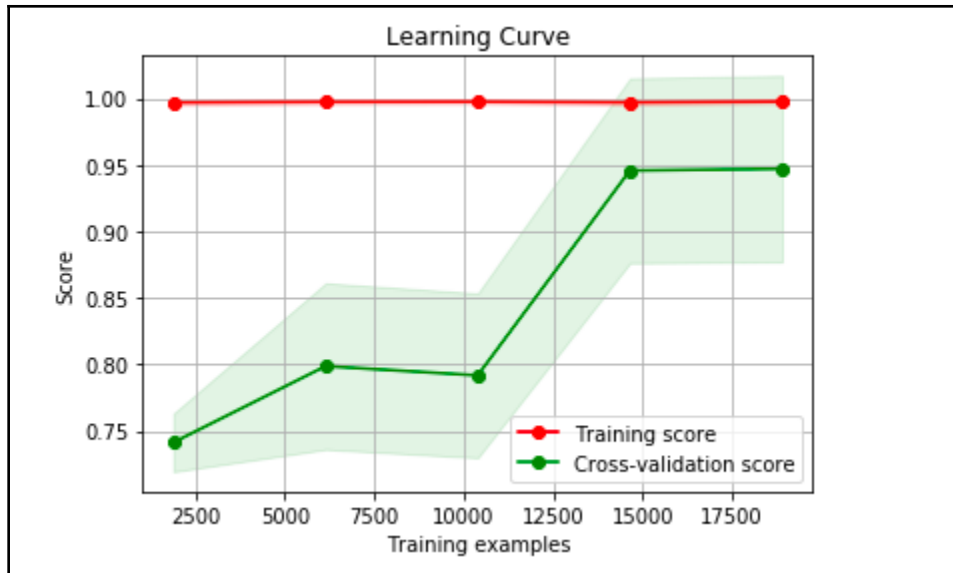
## Learning Curve

The learning curve is a plot that compares how the training accuracy scores and the test accuracy scores vary as the number of samples/rows added to the data increase. In order to construct the learning curve for the K-Nearest Neighbor model we use the code shown below:

```
skplt.estimators.plot_learning_curve(knn_classifier, features, target)

plt.show()
```

This results in a plot as illustrated in the image below:



Learning curve for the K-NN model

From the curve above:

1. The Training Score and the Test Score only are the highest when the number of samples are 15,000. This suggests that even if we had only 15,000 samples instead of the 17,500 we would still get the best possible results.
2. Anything under the 15,000 samples and the test cross-validated scores are much lower than the training scores suggesting that the model is overfit.

## Cross-validated box plot

In this plot, we compare the cross-validated accuracy scores of multiple models by making use of box plots. In order to do this we use the code shown below:

```
from sklearn import model_selection

#List of models

models = [('k-NN', knn_classifier), ('LR', logistic_regression)]

#Initializing empty lists in order to store the results
cv_scores = []
model_name_list = []
```

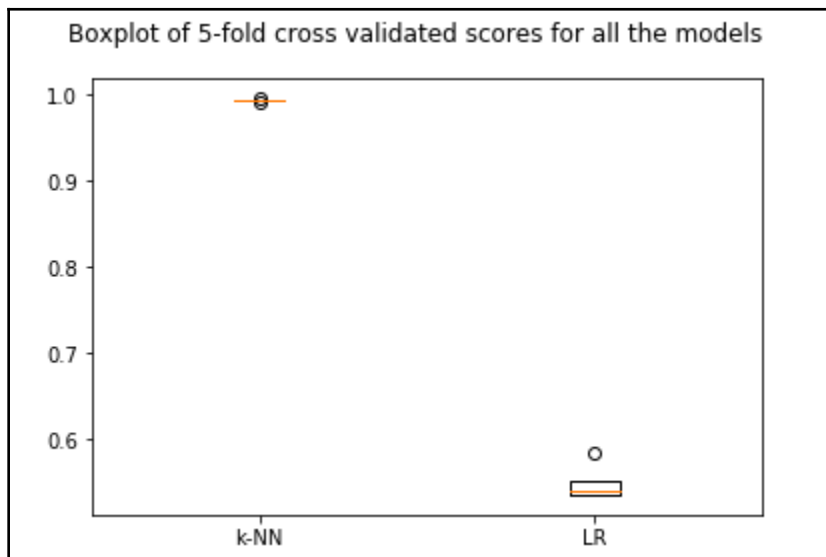
```

for name, model in models:
    #5-fold cross validation
    cv_5 = model_selection.KFold(n_splits= 5, random_state= 50)
    # Evaluating the accuracy scores
    cv_score = model_selection.cross_val_score(model, X_test, y_test, cv =
cv_5, scoring= 'accuracy')
    cv_scores.append(cv_score)
    model_name_list.append(name)
# Plotting the cross-validated box plot

fig = plt.figure()
fig.suptitle('Boxplot of 5-fold cross validated scores for all the models')
ax = fig.add_subplot(111)
plt.boxplot(cv_scores)
ax.set_xticklabels(model_name_list)
plt.show()

```

This results in a plot as shown in the figure below:



Cross validated box plot

In the code above:

1. We first store the models that we want to compare in a list.
2. We then initialize two empty lists in order to store the results of the cross validated accuracy scores and the names of the models so that we can use it later in order to create the box plots.

3. We then iterate over each model in the list of models and use the `model_selection.KFold()` function in order to split the data into 5 a fold cross validated set.
4. Next, we extract the 5 fold cross validates scores by using the `model_selection.cross_val_scores()` function and append the scores along with the model names into the lists we initialized at the beginning of the code.
5. Finally, a box plot is created that displays the cross validated scores in a box plot.

The lists we created fund

amentally consists of the 5 cross validated scores along with the model names. A box plot takes these 5 scores for each model and computes the min, max, median, 1st and 3rd quartile in the form of a box plot.

From the plot:

1. It is clear that the K-NN model has the highest value of accuracy with the lowest difference between the minimum and maximum values.
2. The logistic regression model on the other hand has the highest difference between minimum and maximum values and has an outlier in it's accuracy score as well.

## Performance evaluation for regression algorithms

There are three main metrics that you can use in order to evaluate the performance of the regression algorithm that you built. They are:

1. Mean Absolute Error (MAE)
2. Mean Squared Error (MSE)
3. Root Mean Squared Error (RMSE)

In this section we will learn what the three metrics are, how they work and how you can implement them using scikit-learn.



## Mean Absolute Error (MAE)

The Mean Absolute Error is given by the formula shown below:

$$\frac{1}{n} \sum_{i=1}^n |y_i - \hat{y}_i|$$

MAE Formula

In the formula above, the  $y_i$  represents the true or actual value of the output while  $\hat{y}_i$  represents the predicted output values. Therefore, by computing the summation of the difference between the true value and the predicted value of the output for each row in your data and then dividing it with the total number of observations we get the mean value of the absolute error.

In order to implement the the MAE in scikit-learn we use the code shown below:

```
from sklearn import metrics

metrics.mean_absolute_error(target, predictions)
```

In the code above, the `mean_absolute_error()` function from the metrics module in scikit-learn is used to compute the MAE. It takes in two arguments - the real/true output which is the "target" and the "predictions" which is the predicted outputs.

## Mean Squared Error (MSE)

The Mean Squared Error is given by the formula shown below:

$$\frac{1}{n} \sum_{i=1}^n (y_i - \hat{y}_i)^2$$

Mean Squared Error

The formula above is similar to the formula that we saw for the Mean Absolute Error that we saw earlier except instead of computing the absolute difference between the true and predicted output values we compute the square of the difference.

In order to implement the MSE in scikit-learn we use the code shown below:

```
metrics.mean_squared_error(target, predictions)
```

We use the `mean_squared_error()` function from the `metrics` module with the real/true output values and the predictions as arguments. The Mean Squared Error is better at detecting larger errors because we square the errors instead of only depending on the difference.

## Root Mean Squared Error (RMSE)

The Root Mean Squared Error is given by the formula below:

$$\sqrt{\frac{1}{n} \sum_{i=1}^n (y_i - \hat{y}_i)^2}$$

The formula above is very similar to that of the Mean Squared Error except for the fact that we take the square root of the MSE formula.

In order to compute the RMSE in scikit-learn we use the code shown below:

```
import numpy as np

np.sqrt(metrics.mean_squared_error(target, predictions))
```

In the code above, we use the `mean_squared_error()` function with the true/real output and the predictions and then take the square root of this answer by using the `np.sqrt()` function from the `numpy` package.

Compared to the MAE and the MSE, the RMSE is the best possible metric that you could use in order to evaluate the linear regression model since this detects large errors and gives you value in terms of the output units. The key takeaway from using any one of the 3 metrics is that the value that these metrics give you should be as low as possible, indicating that the model has relatively low error values.

## Performance evaluation for unsupervised algorithms

In this section you will learn how to evaluate the performance of an unsupervised machine learning algorithm such as the K-Means. The first step is to build a simple K-Means model.

We can do this by using the code shown below:

```
#Reading in the dataset

df = pd.read_csv('fraud_prediction.csv')

#Dropping the target feature & the index

df = df.drop(['Unnamed: 0', 'isFraud'], axis = 1)

#Initializing K-means with 2 clusters

k_means = KMeans(n_clusters = 2)
```

Now that we have a simple K-Means model with 2 clusters we can proceed to evaluate the model's performance. The different visual performance charts that can be deployed are as follows:

1. Elbow Plot
2. Silhouette Analysis Plot

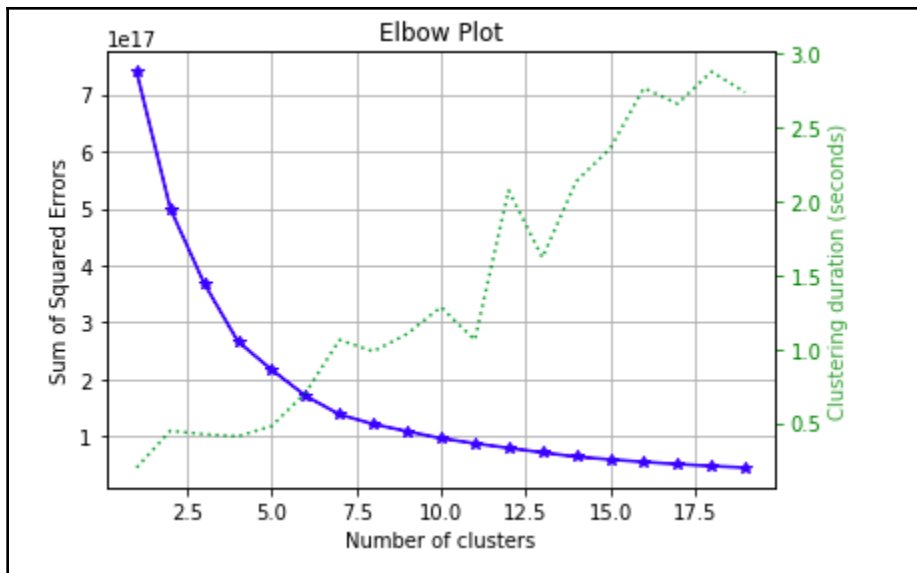
In this section you will learn how to create and interpret each one of the two plots mentioned above.

## Elbow plot

In order to construct an elbow plot we use the code shown below:

```
skplt.cluster.plot_elbow_curve(k_means, df, cluster_ranges=range(1, 20))
plt.show()
```

This results in a plot as illustrated in the image shown below:



Elbow plot

The elbow plot is a plot between the number of clusters the model takes into consideration along the x-axis and the sum of the squared errors along the y-axis.

In the code above:

1. We use the `plot_elbow_curve()` function with the K means model, the data and the number of clusters you want to evaluate.
2. In this case, we define a range of 1 to 19 clusters.

From the plot above:

1. It is clear that the "elbow point" or the point at which the sum of the squared errors (y-axis) starts decreasing very slowly is when the number of clusters is 4.
2. The plot also gives you another interest metric on the y-axis (right hand side) which is the Clustering Duration (in seconds). This tells you about the amount of time it took for the algorithm to create the clusters in seconds.

## Summary

Let's have a quick recap about the concepts that you have learnt in this chapter. You learnt about how you can go about evaluating the performance of the three different types of machine learning algorithms - classification, regression & unsupervised.

For the classification algorithms you learnt how to evaluate the performance of a model by using a series of visual techniques such as the - Confusion Matrix, Normalized Confusion Matrix, Area Under the Curve, KS Statistic Plot, Cumulative Gains Plot, Lift Curve, Calibration Plot, Learning Curve and the Cross-Validated Box Plot.

For the regression algorithms you learnt how to evaluate the performance of a model by using 3 metrics - Mean Squared Error (MSE), Mean Absolute Error (MAE) & the Root Mean Squared Error (RMSE).

Finally, for the unsupervised machine learning algorithms you learnt how to evaluate the performance of a model by using the Elbow plot.

Congratulations! You have now made it to the end of your machine learning journey with scikit-learn. You've made your way through 8 chapters that have given you the quickest entry point into the wonderful world of machine learning with the one of the world's most popular machine learning frameworks - scikit-learn.

In this book you learnt about:

- What machine learning is in a nutshell & the different types & applications of machine learning.
- Supervised machine learning algorithms such as the K-NN, Logistic Regression, Naive Bayes and Support Vector Machines & Linear Regression.
- Unsupervised machine learning algorithms such as the K-Means.
- Algorithms that can perform both classification & regression such the Decision Trees, Random Forests & Gradient Boosted Trees.

I hope you can make the best use of the application based knowledge that this book as given you & solve many real world problems using machine learning as your tool!

# Index