# Build and Evaluate a Predictive Model with scikit-learn:

# Regression

For those new to data science, phrases like predictive modelling and model evaluation may seem complex. Of course depending on the model, these terms may mean less or more complex things. We will attempt to disillusion the perceived complexity of these terms with a guided walkthrough of building a ***Simple Linear Regression (SLR)***and providing an appetizer for evaluating the model with scikit-learn. We will built a conceptual and practical understanding of what predictive modelling is and how to do it.

## **So What is A Prediction Model?**

A prediction model is a function which takes a feature or a set of features **X**as input and outputs some approximation of feature **y** which we wish to predict (y is often referred to as the target feature). This output is an approximation because no model is perfect and therefore our function can only **predict y**within someerror of what the **true y** is. Let’s refer to our predicted approximations of y as **y\_pred** and the true values of y as **y\_true**.

## **Evaluating a Prediction Model: the What, the Why and the How**

In order to evaluate how well a prediction model is performing we need a way of measuring how far or close its predictions of y are from true y. Needless to say, this means that we must have both **y\_pred** and **y\_true**. Why do we care to evaluate our model? Well, because we want to understand whether we can rely on it to make accurate predictions about some future y using new data that it has not yet seen.

For example, in a SLR setting, you can evaluate how good the predictions are by looking at the **mean squared error (MSE) .**This is a metric which takes all the errors our model produced, squares them and then takes the mean of those squared errors. In other words, MSE measures how far our **y\_pred**deviates from **y\_true** on average**.**There are a variety of metrics we can use and MSE is just one of them.

Another metric, for example, is the **coefficient of determination —**the **R\_squared**value**.** R\_squared value measures the amount of variability in the data that is explained by our model as compared to a null model which would simply be always predicting the mean of y. The range for an R\_squared value is between 0 and 1. An R\_squared value of 0.35, for example, would mean that 35% of the variability in our data is explained by our model. The closer the R\_squared value is to 1, the better the model.

To build and evaluate a model, we need to have historical data with which to ‘inform’ our model so it can later make ‘educated’ predictions. By historical data I mean data that includes X and y from the past which we currently have. We feed a subset of this historical data to the model, let it train on it and learn from it. This subset is referred to as **training data**. We then test our model to see how well it’s performing by computing the MSE or the R\_squared value on training data.

But the point of modelling here is not to get the optimal MSE or a great R\_squared value for our training data — we already know what we want our model to predict because our historical and therefore, training data comes with the true y! Instead…

**…we want our model to work really well on ‘future’ data which it has never seen before. This means that we wish to build a model that can be generalizable to new, unseen data.**

Generalizability is an important concept for the predictive power of a model and is achieved by balancing out **the bias and variance trade off**.

## **1. Define X and y**

Let’s say we have data on the rate of cricket chirps (X) and the average outside temperature in Fahrenheit (y). This is our historical data. Keeping in mind the generalizability concept, our goal is to build a model which will predict the outside temperature in the future assuming we will have new data on the rate of cricket chirps — that is, assuming we will have only some new X, but not y at some point in the future.

We define our historical X and y in pandas as follows:

X = data[[‘chirps per second’]]

y = data[‘temperature (F)’]

The double brackets around ‘chirps per second’ are there because we will later be ingesting our X and y into scikit-learn. **Scikit-learn can take any number of predictor features X and it expects a 2-D array for X which is why we need the double brackets**.

## **2. Train-Test Split the Data**

Next we need to partition our historical data into a train set and a test set. This is because we want to train our model, make it smarter, and then we want to test it later to see how well it’s performing.

Let’s define some terminology:

***Train Set*** — a subset of our entire historical dataset which is used to train and build the model.

***Test Set*** — a subset of the entire historical dataset which is used after we have trained our model for the purpose of making predictions and evaluating model performance.

Scikit-learn’s *test\_train\_split* method will randomly split our data into train and test subsets.

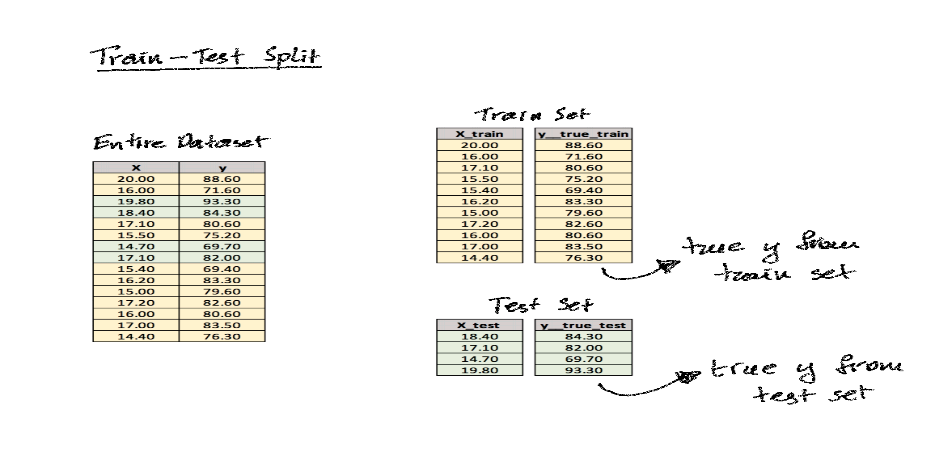
from sklearn.model\_selection import train\_test\_split

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, random\_state=42, test\_size = 0.3)

It’s important to set the *random\_state* parameter to a fixed value. This will ensure that you will get the same train and test sets every time you do a train-test split on the same dataset.

The test\_size parameter here is set to 0.3. This means that I want my test data to be about 30% of the entire dataset, while the remaining portion will become the train set.

The picture below shows what the train and test split accomplishes using a random subset of the actual data used for this example.



Now we’re ready to train and build our model.

## **3. Instantiate the Model**

First step towards actually building our SLR model is to **instantiate**it. This creates a blank model object which is much like an empty template ready to be filled with our training data.

from sklearn.linear\_model import LinearRegression

slr\_model = LinearRegression()

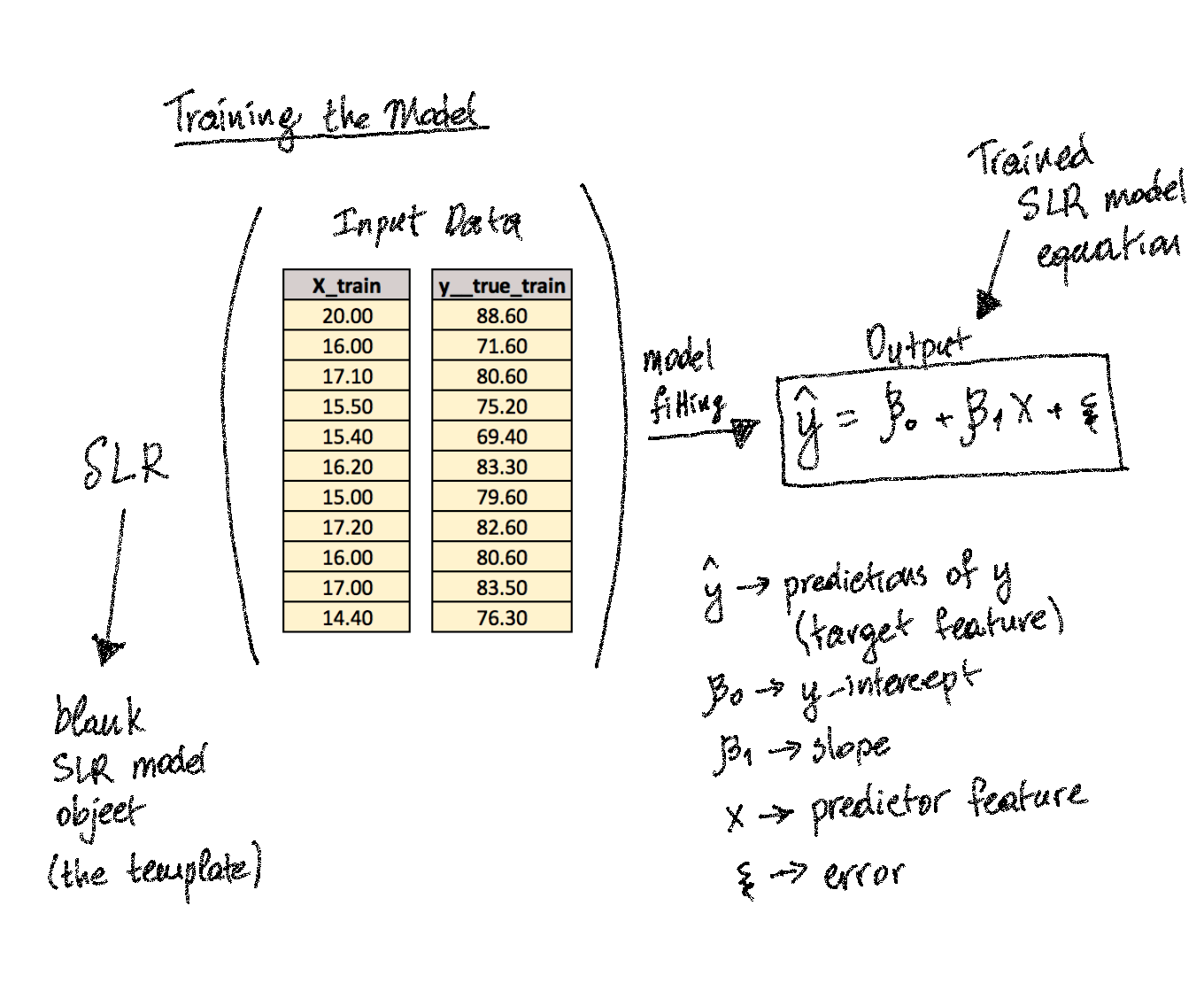
## **4. Fit the Model on Train Set**

Next, we feed our train data into the slr\_model template and let the model ‘fit itself’ to the training data. This is the step where the model is ‘learning’. What this means is that the template slr\_model learns the intricacies of our data and it adjusts itself to it so that when the model sees data like this in the future, it will recognize the most salient patterns in the data and will make its predictions accordingly in a more ‘informed’ way.

**Fitting the model requires inclusion of both X and y from our training set.**

Thanks to scikit learn, this process is a simple one-liner:

slr\_model.fit(X\_train, y\_train)

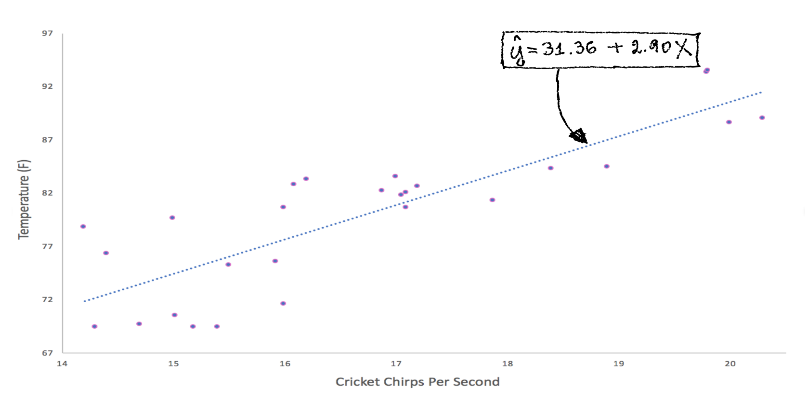


The picture above illustrates the high-level process of what happens when you fit a model. The equation represents the regression line which fits our data and it has unique slope and intercept values. Now let’s look up the slope and the y-intercept for our model and then view a scatterplot with the actual regression line.

slope = slr\_model.coef\_  
y\_intercept = slr\_model.intercept\_

---------------------------------------------------------------------

slope = 2.90165464  
y\_intercept = 31.359529272367354



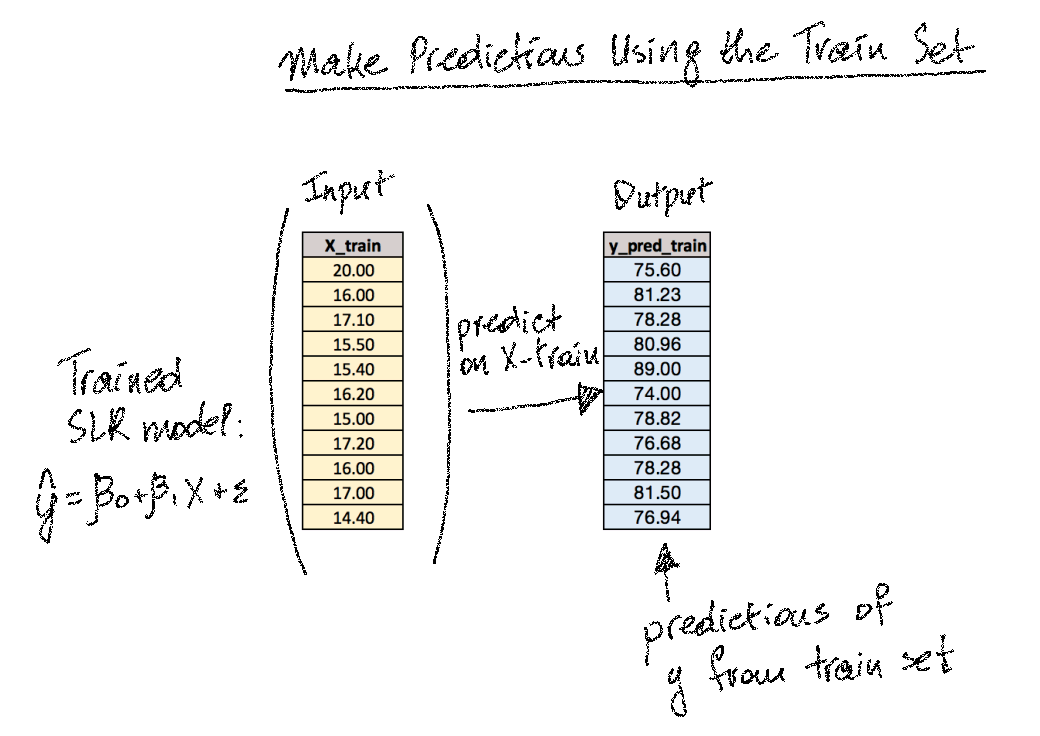
Great! We have a working model! Let’s see how well of a fit it is to our data.

## **5. Make Predictions on Train Set**

In simple terms, the prediction step is equivalent to plugging in X values to our equation from above and calculating the corresponding y values.

The following code produces predictions **y\_pred\_train** on the train set

y\_pred\_train = slr\_model.predict(X\_train)



## **6. Evaluate the Fit on Train Set**

Remember that we have our true y from the train set — **y\_true\_train**. We can compare it to y\_pred\_trainby making use of the R\_squared metric and the MSE metric.

from sklearn.metrics import r2\_score, mean\_squared\_error

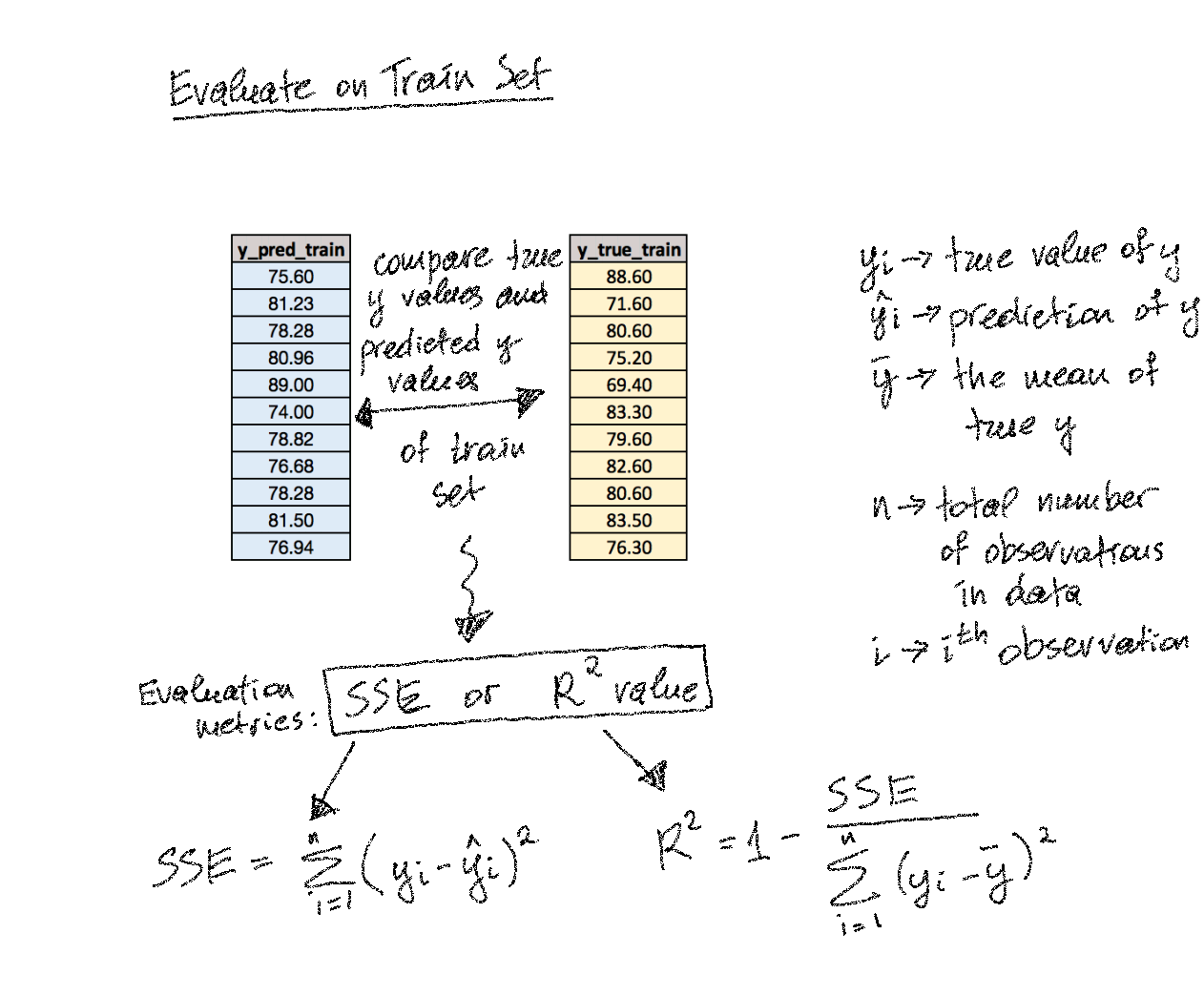
r2\_train = r2\_score(y\_train, y\_pred\_train)

mse\_train = mean\_squared\_error(y\_train, y\_pred\_train)

----------------------------------------------------------------------------------------------------------------------

[Output]

r2\_train = 0.6599654904754202  
mse\_train = 12.744526804119651



The training R\_squared value is 0.66. This means that the model we just built explains about 66% of the variability in our data.

The training MSE is 12.74 which means our predictions on average deviate from the true values of y by about 3.57 degrees of Farenheit. The MSE is the mean **squared** error, so in order to get the actual error with the same units as our y (deg. Farenheit) we need to take the square root of the MSE.

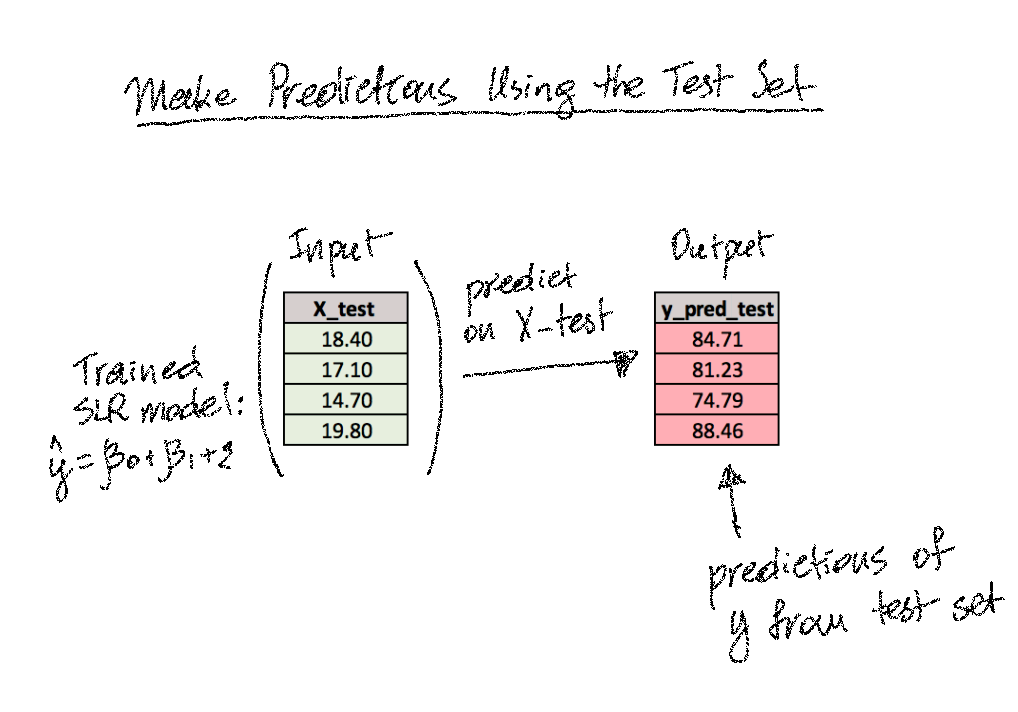
**It’s critical to understand the purpose of the train set and the test set.**We train and build the model using the train set. We can evaluate the model on the train set to see how well the fit is. But we use the test set to gain insight on how generalizable the model is to future data. We only make predictions on X\_test and never ingest y\_true\_test to the model.

**We always want to make sure our model is agnostic to y\_test\_true because y\_test\_true represents unseen data.**

## **7. Make Predictions on Test Set**

Just like we saw before with predicting on train data, we can do the same on test data:

y\_pred\_test = slr\_model.predict(X\_test)



## **8. Evaluate Performance on Test Set**

And our test scores are…

r2\_test = r2\_score(y\_test, y\_pred\_test)  
mse\_test = mean\_squared\_error(y\_test, y\_pred\_test)

----------------------------------------------------------------[Output]---------------------------------------------------------------

r2\_test = 0.7728120770684193  
mse\_test = 16.095126545492537

The testing scores can be interpreted in the same way as we did the training scores above.

We now have both our training scores and the testing scores which means we can compare the two. By comparing the performance of the model on train set to that of the test set, we can infer a lot about how good our model really is.

**Types of Logistic Regression:**

* Binary Logistic Regression: The target variable has only two possible outcomes such as Spam or Not Spam, Cancer or No Cancer.
* Multinomial Logistic Regression: The target variable has three or more nominal categories such as predicting the type of Wine.
* Ordinal Logistic Regression: the target variable has three or more ordinal categories such as restaurant or product rating from 1 to 5.

## Model building in Scikit-learn

Let's build the diabetes prediction model.

Here, you are going to predict diabetes using Logistic Regression Classifier.

Let's first load the required Pima Indian Diabetes dataset using the pandas' read CSV function. You can download data from the following link: <https://www.kaggle.com/uciml/pima-indians-diabetes-database>

#### Loading Data

#import pandas

import pandas as pd

# load dataset

pima = pd.read\_csv('/Users/user/Desktop/7BUIS008W/diabetes.csv')

pima.head()

#### **Selecting Feature**

Here, you need to divide the given columns into two types of variables dependent(or target variable) and independent variable(or feature variables).

#### #split dataset in features and target variable

#### feature\_cols = ['Pregnancies', 'Insulin', 'BMI', 'Age','Glucose','BMI','DiabetesPedigreeFunction']

#### X = pima[feature\_cols] # Features

#### y = pima.Outcome # Target variable

#### **Splitting Data**

To understand model performance, dividing the dataset into a training set and a test set is a good strategy.

Let's split dataset by using function train\_test\_split(). You need to pass 3 parameters features, target, and test\_set size. Additionally, you can use random\_state to select records randomly.

from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()

X1 = scaler.fit\_transform(X)

# split X and y into training and testing sets

from sklearn.model\_selection import train\_test\_split

X1\_train,X1\_test,y\_train,y\_test=train\_test\_split(X1,y,test\_size=0.25,random\_state=0)

Here, the Dataset is broken into two parts in a ratio of 75:25. It means 75% data will be used for model training and 25% for model testing.

#### **Model Development and Prediction**

First, import the Logistic Regression module and create a Logistic Regression classifier object using LogisticRegression() function.

Then, fit your model on the train set using fit() and perform prediction on the test set using predict().

### # import the class

### from sklearn.linear\_model import LogisticRegression

### # instantiate the model (using the default parameters)

### logreg = LogisticRegression()

### logreg.fit(X1\_train, y\_train)

y\_pred=logreg.predict(X1\_test)

### ***Model Evaluation using Confusion Matrix***

A confusion matrix is a table that is used to evaluate the performance of a classification model. You can also visualize the performance of an algorithm. The fundamental of a confusion matrix is the number of correct and incorrect predictions are summed up class-wise.

from sklearn.metrics import accuracy\_score

accuracy = accuracy\_score(y\_test,y\_pred)

print (accuracy)

# import the metrics class

from sklearn import metrics

cnf\_matrix = metrics.confusion\_matrix(y\_test, y\_pred)

cnf\_matrix

---------------------------------------------------------------------------------------------------------------------------------------------------

0.7916666666666666

array([[117, 13],

[ 27, 35]])

Here, you can see the confusion matrix in the form of the array object. The dimension of this matrix is 2\*2 because this model is binary classification. You have two classes 0 and 1. Diagonal values represent accurate predictions, while non-diagonal elements are inaccurate predictions. In the output, 117 and 35 are actual predictions, and 27 and 13 are incorrect predictions.

#### **Visualizing Confusion Matrix using Heatmap**

Let's visualize the results of the model in the form of a confusion matrix using matplotlib and seaborn. Here, you will visualize the confusion matrix using Heatmap.

# import required modules

import numpy as np

import matplotlib.pyplot as plt

import seaborn as sns

%matplotlib inline

class\_names=[0,1] # name of classes

fig, ax = plt.subplots()

tick\_marks = np.arange(len(class\_names))

plt.xticks(tick\_marks, class\_names)

plt.yticks(tick\_marks, class\_names)

# create heatmap

sns.heatmap(pd.DataFrame(cnf\_matrix), annot=True, cmap="YlGnBu" ,fmt='g')

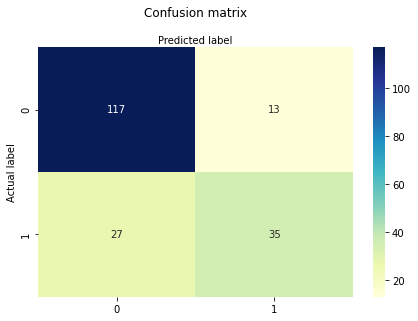
ax.xaxis.set\_label\_position("top")

plt.tight\_layout()

plt.title('Confusion matrix', y=1.1)

plt.ylabel('Actual label')

plt.xlabel('Predicted label')



#### **Confusion Matrix Evaluation Metrics**

Let's evaluate the model using model evaluation metrics such as accuracy, precision, and recall.

print("Accuracy:",metrics.accuracy\_score(y\_test, y\_pred))

print("Precision:",metrics.precision\_score(y\_test, y\_pred))

print("Recall:",metrics.recall\_score(y\_test, y\_pred))

=======================================================================================

Accuracy: 0.8072916666666666

Precision: 0.7659574468085106

Recall: 0.5806451612903226

Well, you got a classification rate of 80%, considered as good accuracy.

Precision: Precision is about being precise, i.e., how accurate your model is. In other words, you can say, when a model makes a prediction, how often it is correct. In your prediction case, when your Logistic Regression model predicted patients are going to suffer from diabetes, that patients have 76% of the time.

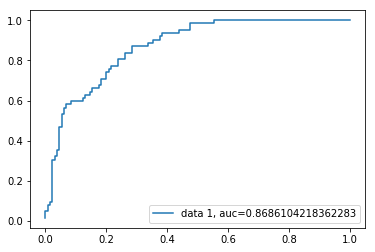
Recall: If there are patients who have diabetes in the test set and your Logistic Regression model can identify it 58% of the time.

#### **ROC Curve**

Receiver Operating Characteristic(ROC) curve is a plot of the true positive rate against the false positive rate. It shows the tradeoff between sensitivity and specificity.

from sklearn.metrics import plot\_roc\_curve

plot\_roc\_curve(logreg, X1\_test, y\_test)



AUC score for the case is 0.86. AUC score 1 represents perfect classifier, and 0.5 represents a worthless classifier.

### ***L-Reg Advantages***

Because of its efficient and straightforward nature, doesn't require high computation power, easy to implement, easily interpretable, used widely by data analyst and scientist. Also, it doesn't require scaling of features. Logistic regression provides a probability score for observations.

### ***L-Reg Disadvantages***

Logistic regression is not able to handle a large number of categorical features/variables. It is vulnerable to overfitting. Also, can't solve the non-linear problem with the logistic regression that is why it requires a transformation of non-linear features. Logistic regression will not perform well with independent variables that are not correlated to the target variable and are very similar or correlated to each other.

### ***Conclusion***

In this tutorial, we covered a lot of details about predictive modelling and Regression. You have learned what the line and logistic regression is, how to build respective models, how to visualize results and some of the theoretical background information. Also, you covered some basic concepts, confusion matrix, ROC curve.