# Build and Evaluate Predictive Models with Scikit-Learn

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 ***Logistic Regression*** *(LR)*, providing an appetiser for evaluating the model with scikit-learn. We will build a conceptual and practical understanding of what predictive modelling is and how to do it. But first let’s start with some foundation concepts.

**Part (1) Understanding Training, Testing and Predicting**

A predictive model is build using historical data, this is called ***TRAINING.*** A goal of a predictive model (supervised learning) is to build a model that performs well on new data from real world, this data which the model did not learn before (unseen data), this assessment of model on unseen data is called ***TESTING.*** If you have new data, it’s a good idea to see how your model performs on it. The problem is that you may not have new data, getting new additional data from the real world is expensive, additional real-world data costs both time and money. Therefore, there is a workaround, we can pretend to have new unseen data! Or in other words, you can simulate this experience (having new data) with a procedure like **Train Test Split**.

**A. Training and Testing**

When building a machine learning model, we use a dataset known as the training dataset to train the algorithm. However, to understand how good is the fitting of the built model, we must test it on new data from the real world which was not used in training (unknown to the model).

* Remember that the original dataset has a number of **input features** known as***Independent Variables (IV)***, these are called the **X-Columns.** The original dataset also has one **output feature** known as ***Dependent Variable (DV)***, this is known as the **y-Column.** See **fig.1** below:

A table with numbers and a black background

Description automatically generated

**Fig.1** Illustration of Independent Variables (IV) and the Dependent Variable (DV)

* To pretend that we have new data for testing, before training the algorithm, we intentionally **remove/hide** away some of the records (the rows AKA instances) from the original dataset. We group these *isolated (hidden) subset records* aside and pretend that these records are a new dataset for to test the predictive model after it is built, this is the **Test Set.** The algorithms must never learn the data in the test set. The remining records only can be used for building the model **(training the algorithm)**, these remaining records are called the **Training set**. See **fig.2** below.

A table with numbers and a black background

Description automatically generated

**Fig.2** Illustration of creating a training and test datasets

Congratulations! You have a Training set and a Test set now!

* Remember the original data had **X-Columns** and a **Y-Column**. Therefore, the Training set will have its own **X-Columns** and a **y-Column**. Since both belong to the Training set these are called **X\_Train** and **y\_Train.**

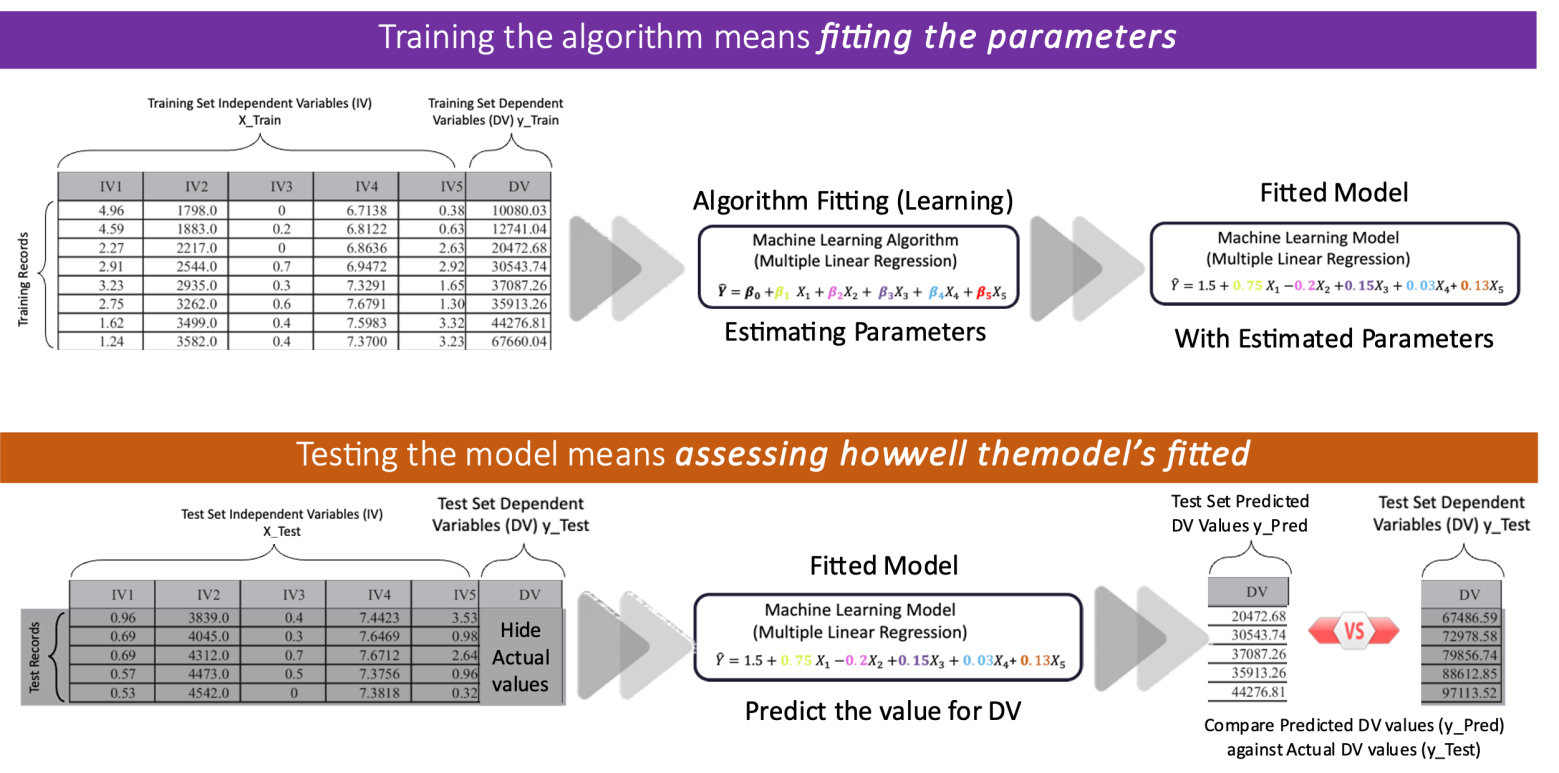
Similarly,the test set will have its own **X-Columns** and a **y-Column**. Since both belong to the Test set these are called **X\_Test** and **y\_Test.** See **fig.3** below.

A table with numbers and a black background

Description automatically generated

**Fig.3** Illustration of training test split operation

* Training the algorithm estimates its parameters to create a model. The test set assesses the model if it produces **predicted output values (y\_Predicted** or shorted **y\_Pred)** that are the same of close to the **actual output** **values** **(y\_Test)** which we have hidden. We evaluate the model by comparing y\_Pred values produced by machine learning to y\_Test value which are the actual values recorded for y in the test dataset (See fig.4 ).



**Fig.4** Illustration of the purpose of training test split operation

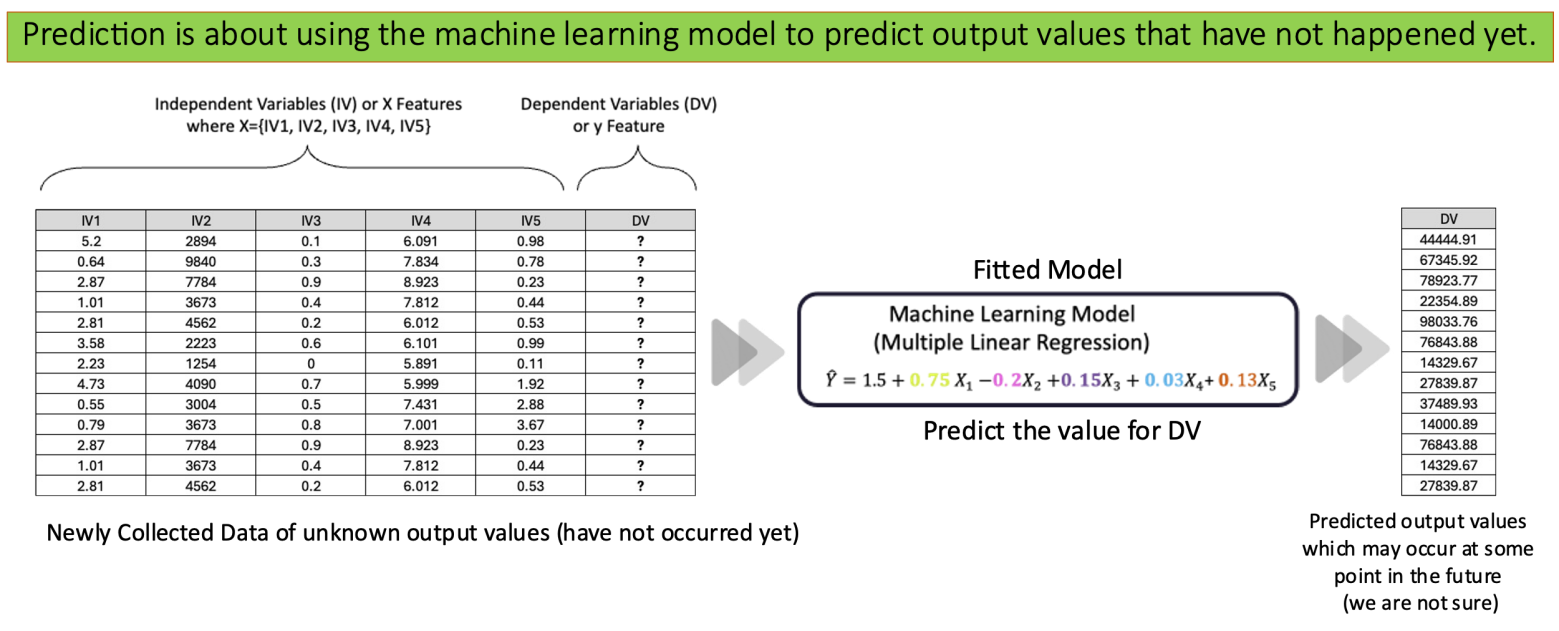
* To perform this comparison, we use simple mathematics. For one instance (individual or row) with a numeric output variable, if the predicted output value is 5 and the actual output value (recorded in the test set) is 9, then the error is the difference 4. This is mathematically expressed as:

Therefore, if the Error is ZERO, this means there is no difference between the predicted and actual output values, the prediction is accurate in this case. Depending on the type of output values, we can calculate performance metrics such as **Mean Square Error (MSE), Mean Absolute Error (MAE), coefficient of determination***(***R2), Accuracy, Recall, Precision** and others. *(No need to worry about these now, they will be discussed in detail in lecture 4)*

**B. Predicting**

This is the stage when we deploy machine learning in the real-world to estimate future values for us that have not happened yet! But we only deploy the model if stake holders are satisfied with its test performance. This satisfaction is translated into **Meeting Success Criteria** for the machine learning project.

* After assessing the model, if the business stakeholders are satisfied with its **Test** performance, this model can be deployed in real life **to predict new values**. Unlike testing, the model will be given a new set of input features values (measurements) for an individual (instance), which at that point of time the output has not happened (is truly unknown), thus, the machine learning model will try to predict the output value for them. These predicted values by machine learning have not yet occurred but they will certainly occur sometime in the future. (See fig. 5)
* For example, machine learning can predict if a person may die of cancer in the future after a number of months while the person is still alive now. This prediction can be inaccurate or accurate to a degree.



**Fig.5** Illustration of using machine learning for predictions on real-world data

**C. Partitioning the data into Training and Test sets in SK-Learn**

Partitioning (splitting) our historical data into a train set and a test set means the following:

**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 to evaluate the model’s 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 **70%** becomes the train set.

**Part (2) Regression Predictive Model – Can crickets tell the temperature?**

We all know the evening sound of a cricket. Did you know that the number of cricket chirps reports the temperature? It’s true! **Let’s use a Cricket as a Thermometer!**

A black insect on the ground

Description automatically generated

Crickets are cold-blooded and take on the temperature of their surroundings. In 1897, a scientist named **Amos Dolbear** published an article titled “The Cricket as a Thermometer” that noted the correlation between the ambient temperature and the rate at which crickets chirp. The insects’ muscles contract to produce chirping based on chemical reactions. The warmer the temperature, the easier the cricket’s muscles activate, so the chirps increase. The cooler the temperature, the slower the reaction rate, and the less frequent the chirps are, the lower the chirp rate.

*Chirping is a cricket’s way of communicating. Male crickets use chirping to attract females, scare off other males, or warn of danger. <https://youtube.com/shorts/fFL432fktwE?si=BvTbt-M-0-_VfUIT>*

**TASK: Build SLR model to predict outdoor temperature based on chirps’ rate**

1. Load your temperature and chirps rate dataset (slr.csv) and access the data values in Colab.

*Code cell:*

import pandas as pd

data=pd.read\_csv('/content/slr.csv')

data.head()

*Output cell:*

A screenshot of a graph

Description automatically generated

1. From our historical data we see the recorded values of chirps per second and the corresponding measured outdoor temperature in Fahrenheit. Our goal is to build a model which will predict the outside temperature in the future.

We define our historical input feature **X : the chirps per second** and the output feature **y: temperature (F)** in pandas as follows:

*Code cell:*

X = data[['chirps per second']]

y = data['temperature (F)']

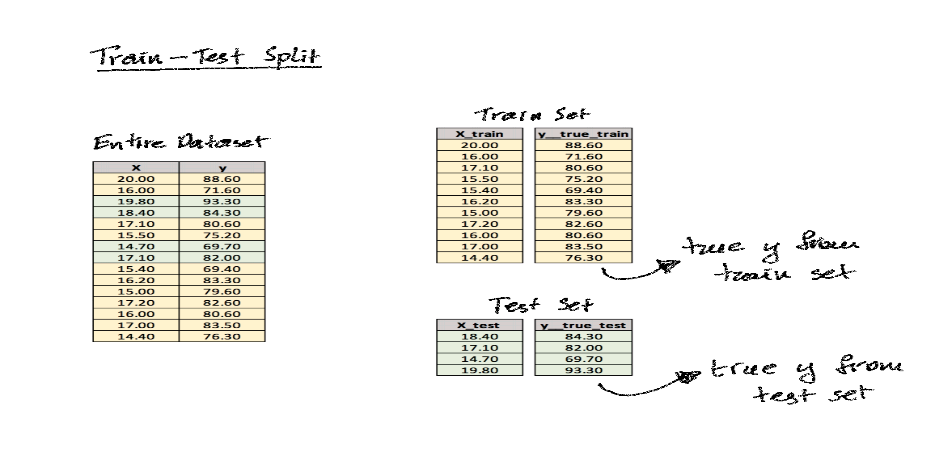
**Note:** The double brackets around *‘chirps per second’*arethere because we will later be ingesting our X and y into scikit-learn. Scikit-learn can take any number of predictor features X and it expects a 2-D array for X which is why we need the double brackets.

1. 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 (see fig.6)

*Code cell:*

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 = 0.25)



**Fig.5** Illustration of splitting the slr dataset into training and test sets

1. You can check the size of your whole data and the size of training and test sets using the shape function.

*Code cell:*

print('Whole Data shape', data.shape)

print('X\_train shape', X\_train.shape)

print('X\_test shape', X\_test.shape)

*Output cell:*

A black background with white text

Description automatically generated

1. **Instantiate the Model:** First step towards building our Single Linear Regression (SLR) model is to *instantiate*it. This creates a blank model object (algorithm) which is much like an empty template ready to be filled (fitted) with our training data.

*Code cell:*

from sklearn.linear\_model import LinearRegression

slr\_model = LinearRegression()

1. **Fit the Model on Train Set:** 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. In other words, the parameters values for this algorithm are estimated at this stage to create a model.

*Code cell:*

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

1. Now let’s look up the slope and the y-intercept for our model and then view a scatterplot with the actual regression line.

*Code cell:*

#To see the simple linear regression parameters

slope = slr\_model.coef\_

y\_intercept = slr\_model.intercept\_

print('Slope', slope)

print('Intercept', y\_intercept)

*Output cell:*

A close up of numbers

Description automatically generated

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

1. **Make Predictions on Train Set:** This prediction step is equivalent to plugging in X values to our equation from above and calculating the corresponding y values. But this is on the training dataset, this is done assess how good the learning on the training data.

*Code cell:*

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

1. To evaluate the model’s learning, we compare our actual **y** from the train set **(*y\_train)*** *to the* ***predicted values*** *of* ***y*** *(* *y\_pred\_train)*by making use of the **R2** metric andthe **MSE metrics** (to be discussed in lecture 4)**.** Here we are asking the question **How well did the model learn the training data?**

*Code cell:*

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 cell:*

A black background with white numbers

Description automatically generated

The training **R2** value is 0.70. This means that the model we just built explains about 70% of the variability in our data. The training **MSE** is 13.77 which means our predictions on average deviate from the true values of y by about 3.71 degrees of Fahrenheit. The MSE is the mean ***squared*** error, so to get the actual error with the same units as our **y** (deg. Fahrenheit) we need to take the **square root of the MSE**.

10. **Make predictions on the test data:** It’s critical to understand the purpose of the train set and the test set. We use the test set to gain insight on how generalisable the model is to future data. Here we are asking the question **How well is the model going to perform if I use it to make prediction on new unseen data?**

*Code cell:*

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

11. **Evaluate the model’s performance on Test Set:** use the same performance metrics, the **R2** metric and **the MSE**

*Code cell:*

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

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

print('Test R2 score', r2\_test)

print('Test MSE', mse\_test)

*Output cell:*

A black background with white numbers

Description automatically generated

12. To compare the actual values of temperature to the predicted values of temperature for the test data, you can create a data frame to compare the test instances **y\_test** and **y\_pred\_test** in **comparison\_df.** You can also save the table as a .csv file. Pay attention to the indexes of the instances in your test set and compare them to the student next to you. Observe the differences between the actual values of recorded temperature and the predicted values.

*Code cell:*

Comparison\_df = pd.DataFrame({'Actual' : y\_test, 'Predicted' : y\_pred\_test})

Comparison\_df.to\_csv(r'/content/Comparison\_df.csv', index=True)

Comparison\_df

*Output cell:*

A screenshot of a graph

Description automatically generated

*Output cell:*

A screen shot of a computer

Description automatically generated

**Part (3) Classification Predictive Models – Can we predict diabetes status?**

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>

There are multiple types of Logistic regression algorithms. **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.

**TASK: Build a Logistic Regression (LR) model to predict patient’s diabetes diagnoses**

1. Load your Pima Indian Diabetes dataset (diabetes.csv) and access the data values in Colab.

*Code cell:*

#import pandas

import pandas as pd

# load dataset

pima = pd.read\_csv('/content/diabetes.csv')

pima.head()

*Output Cell*

A screenshot of a graph

Description automatically generated

1. **Selecting Feature:** you need to divide the given columns into two types of variables dependent(or target variable) and independent variable(or feature variables).

*Code cell:*

#split dataset in features and target variable

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

X = pima[feature\_cols] # Features

y = pima.Outcome # Target variable

1. **Scale your input variables:** notice there isa difference in magnitude in your dataset, this can impact the logistic regression performance. Use standardization to scale your input features.

*Code cell:*

from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()

X1 = scaler.fit\_transform(X)

1. **Splitting Data into training and tests sets:** 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. 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.

*Code cell:*

# 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)

print('Whole Data shape', pima.shape)

print('X1\_train shape', X1\_train.shape)

print('X1\_test shape', X1\_test.shape)

*Output cell:*

***A black background with white text

Description automatically generated***

1. **Instantiate the Model:** The first step towards building our Logistic Regression (LR) classification model is to *instantiate*it. This creates a blank model object (algorithm), which is much like an empty template ready to be filled (fitted) with our training data. Import the **Logistic Regression module** and create a **Logistic Regression classifier object** using the **LogisticRegression() function**. Then, fit your model on the train set using **fit()** and perform prediction on the test set using **predict()**.

*Code cell:*

# 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)

1. To view the predicted values for diabetes for the test dataset, simply view the array of values created by applying the **predict()** function; these are stored in **y\_pred.** Remember, **1** indicates diabetic positive and **0** is diabetic negative.

*Code cell:*

#To see the predicted values

y\_pred

*Output cell:*

A screenshot of a computer code

Description automatically generated

To make a visual comparison between the predicted diabetic diagnoses (results) and the actual diabetic values recorded for each patient in the dataset, create a comparison data frame **Comparison\_df** and save it with its indexes in a .csv file. Observe the mismatches between the actual diabetic diagnosis values recorded for patients and the predicted diagnoses for the same patients.

*Code cell:*

Comparison\_df = pd.DataFrame({'Actual Diabetic Diagnoses' : y\_test, 'Predicted' : y\_pred})

Comparison\_df.to\_csv(r'/content/Diagnoses\_Comparison\_df.csv', index=True)

Comparison\_df

*Output cell:*

A screenshot of a computer

Description automatically generated

In order to view all the predictions with truncation view, you could simply expand the display options in pandas so that you can view your desired number of instances before truncation. Remember, the shape of your test set, it has just under 200 instances. So let’s scale the view to 200 instances.

*Code cell:*

pd.set\_option('display.max\_rows', 200)

pd.set\_option('display.max\_columns', 200)

pd.set\_option('display.width', 200)

*Output cell:*

A screenshot of a computer

Description automatically generated

1. **Model Evaluation using confusion matrix:** There are three types of classification evaluation metrics: **fundamental**, **combined** and **graphical**. A confusion matrix is a fundamental table used to evaluate a classification model's performance. You can also visualise the performance of an algorithm. The fundamental of a confusion matrix is the **number of correct and incorrect predictions** that are summed up class-wise. This table will be discussed in detail in Lecture 4. The **confusion\_matrix** module is imported from the **metrics** package in the **sklearn** library and is the **ConfusionMatrixDisplay**

*Code cell:*

# To plot the confusion matrix

from sklearn.metrics import confusion\_matrix

from sklearn.metrics import ConfusionMatrixDisplay

# Construct the confusion matrix cm

cm = confusion\_matrix(y\_test, y\_pred, labels=logreg.classes\_)

# Create a display to plot the confusion matrix

disp = ConfusionMatrixDisplay(cm,display\_labels=logreg.classes\_)

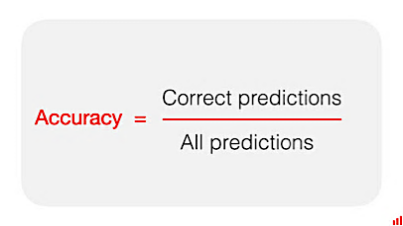
disp.plot()

*Output cell:*

A chart of a blue yellow and purple box

Description automatically generated with medium confidence

1. **The machine learning LR model accuracy:** to calculate the module accuracy, apply the **accuracy\_score** function to compare the actual diagnoses in the test dataset **y\_test** against the predicted diagnoses values by the logistic regression model, **y\_pred.** Test Accuracy is the ratio ofpredictions made by the model that match the actual diagnoses to the total number of instances in the test dataset.



*Code cell:*

# Import the function to calculate accuracy score

from sklearn.metrics import accuracy\_score

# Apply the function to find the correct predictions

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

# Display the accuracy

print ('The Logistic Regression Model Accuracy:',accuracy)

*Output cell:*



1. **The classification metrics report:** This report calculates more sophisticated classification performance metrics known as **combined metrics**. These combined metrics, **Accuracy, Recall, Precision and F-Score,** are calculated from the confusion matrix and displayed in a report. To obtain the classification report, **use the classification\_report** function from the **metrics** package in sklearn. These metrics will be explained in detail during lecture 4.

*Code cell:*

from sklearn.metrics import classification\_report

print(classification\_report(y\_test, y\_pred))

*Output cell:*

***A screenshot of a computer

Description automatically generated***

1. **ROC Curve: *The*** Receiver Operating Characteristic(ROC) curve is a graphical classification metric plot of the true positive rate against the false positive rate. It shows the trade-off between sensitivity and specificity. All these metrics will be explained in detail during lecture 4. To plot the ROC curve, use the **RocCurveDisplay** function from the **metrics** package in **sklearn**.

*Code cell:*

# Import the function from the package

from sklearn.metrics import RocCurveDisplay

# Apply the function by specifying the name of your model and test data.

Logreg\_roc = RocCurveDisplay.from\_estimator(logreg, X1\_test, y\_test)

*Output cell:*

