[Machine Learning A-Z (Codes and Datasets)](https://drive.google.com/drive/folders/1OFNnrHRZPZ3unWdErjLHod8Ibv2FfG1d)

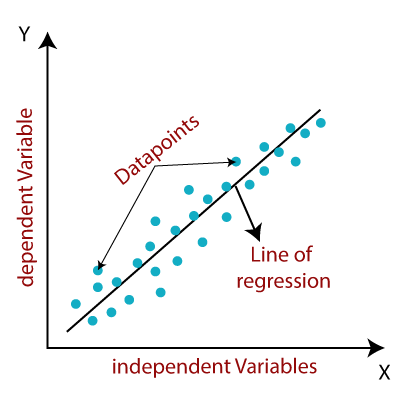
**Practical link**

# Linear Regression in Machine Learning

Linear regression is one of the easiest and most popular Machine Learning algorithms. It is a statistical method that is used for predictive analysis. Linear regression makes predictions for continuous/real or numeric variables such as sales, salary, age, product price, etc.

Linear regression algorithm shows a linear relationship between a dependent (y) and one or more independent (y) variables, hence called as linear regression. Since linear regression shows the linear relationship, which means it finds how the value of the dependent variable is changing according to the value of the independent variable.

The linear regression model provides a sloped straight line representing the relationship between the variables. Consider the below image:



Mathematically, we can represent a linear regression as:

*y= a0+a1x+ ε*

Here,

Y= Dependent Variable (Target Variable)  
X= Independent Variable (predictor Variable)  
a0= intercept of the line (Gives an additional degree of freedom)  
a1 = Linear regression coefficient (scale factor to each input value).  
ε = random error

The values for x and y variables are training datasets for Linear Regression model representation.

## Types of Linear Regression

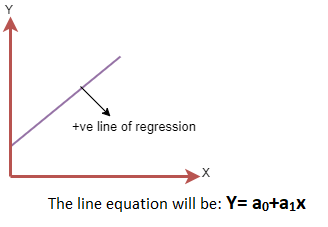
Linear regression can be further divided into two types of the algorithm:

* Simple Linear Regression:  
  If a single independent variable is used to predict the value of a numerical dependent variable, then such a Linear Regression algorithm is called Simple Linear Regression.
* Multiple Linear regression:  
  If more than one independent variable is used to predict the value of a numerical dependent variable, then such a Linear Regression algorithm is called Multiple Linear Regression.

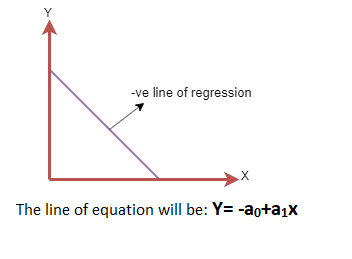
## Linear Regression Line

A linear line showing the relationship between the dependent and independent variables is called a regression line. A regression line can show two types of relationship:

* Positive Linear Relationship:  
  If the dependent variable increases on the Y-axis and independent variable increases on X-axis, then such a relationship is termed as a Positive linear relationship.



* Negative Linear Relationship:  
  If the dependent variable decreases on the Y-axis and independent variable increases on the X-axis, then such a relationship is called a negative linear relationship.



## Finding the best fit line:

When working with linear regression, our main goal is to find the best fit line that means the error between predicted values and actual values should be minimized. The best fit line will have the least error.

The different values for weights or the coefficient of lines (a0, a1) gives a different line of regression, so we need to calculate the best values for a0 and a1 to find the best fit line, so to calculate this we use cost function.

### Cost function-

* The different values for weights or coefficient of lines (a0, a1) gives the different line of regression, and the cost function is used to estimate the values of the coefficient for the best fit line.
* Cost function optimizes the regression coefficients or weights. It measures how a linear regression model is performing.
* We can use the cost function to find the accuracy of the mapping function, which maps the input variable to the output variable. This mapping function is also known as Hypothesis function.

For Linear Regression, we use the Mean Squared Error (MSE) cost function, which is the average of squared error occurred between the predicted values and actual values. It can be written as:

For the above linear equation, MSE can be calculated as:

Linear Regression in Machine Learning

Where,

N=Total number of observation  
Yi = Actual value  
(a1xi+a0)= Predicted value.

# Simple Linear Regression in Machine Learning

Simple Linear Regression is a type of Regression algorithms that models the relationship between a dependent variable and a single independent variable. The relationship shown by a Simple Linear Regression model is linear or a sloped straight line, hence it is called Simple Linear Regression.

The key point in Simple Linear Regression is that the *dependent variable must be a continuous/real value*. However, the independent variable can be measured on continuous or categorical values.

Simple Linear regression algorithm has mainly two objectives:

* Model the relationship between the two variables. Such as the relationship between Income and expenditure, experience and Salary, etc.
* Forecasting new observations. Such as Weather forecasting according to temperature, Revenue of a company according to the investments in a year, etc.

## Simple Linear Regression Model:

The Simple Linear Regression model can be represented using the below equation:

*y= a0+a1x+ ε*

Where,

a0= It is the intercept of the Regression line (can be obtained putting x=0)  
a1= It is the slope of the regression line, which tells whether the line is increasing or decreasing.  
ε = The error term. (For a good model it will be negligible)

## Implementation of Simple Linear Regression Algorithm using Python

Problem Statement example for Simple Linear Regression:

Here we are taking a dataset that has two variables: salary (dependent variable) and experience (Independent variable). The goals of this problem is:

* We want to find out if there is any correlation between these two variables
* We will find the best fit line for the dataset.
* How the dependent variable is changing by changing the independent variable.

In this section, we will create a Simple Linear Regression model to find out the best fitting line for representing the relationship between these two variables.

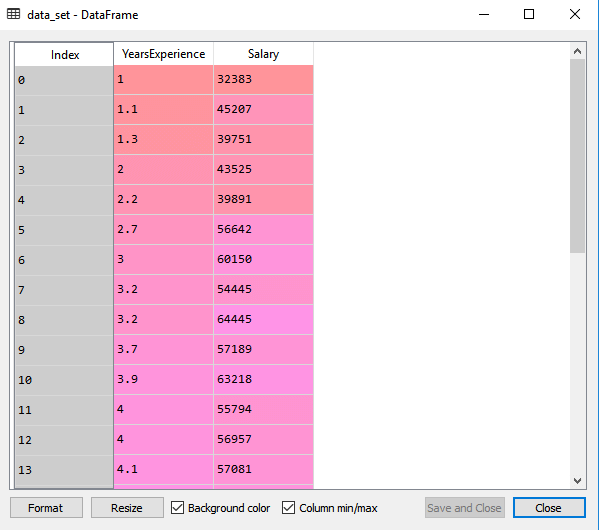
To implement the Simple Linear regression model in machine learning using Python, we need to follow the below steps:

Step-1: Data Pre-processing

The first step for creating the Simple Linear Regression model is [data pre-processing](https://www.javatpoint.com/data-preprocessing-machine-learning). We have already done it earlier in this tutorial. But there will be some changes, which are given in the below steps:

* First, we will import the three important libraries, which will help us for loading the dataset, plotting the graphs, and creating the Simple Linear Regression model.
* **import** numpy as nm
* **import** matplotlib.pyplot as mtp
* **import** pandas as pd
* Next, we will load the dataset into our code:
* data\_set= pd.read\_csv('Salary\_Data.csv')

By executing the above line of code (ctrl+ENTER), we can read the dataset on our Spyder IDE screen by clicking on the variable explorer option.



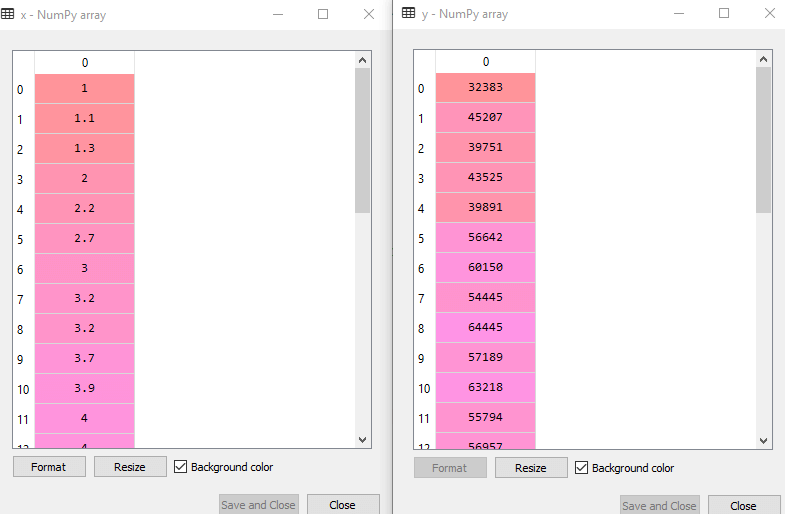
The above output shows the dataset, which has two variables: Salary and Experience.

#### *Note: In Spyder IDE, the folder containing the code file must be saved as a working directory, and the dataset or csv file should be in the same folder.*

* After that, we need to extract the dependent and independent variables from the given dataset. The independent variable is years of experience, and the dependent variable is salary. Below is code for it:
* x= data\_set.iloc[:, :-1].values
* y= data\_set.iloc[:, 1].values

In the above lines of code, for x variable, we have taken -1 value since we want to remove the last column from the dataset. For y variable, we have taken 1 value as a parameter, since we want to extract the second column and indexing starts from the zero.

By executing the above line of code, we will get the output for X and Y variable as:

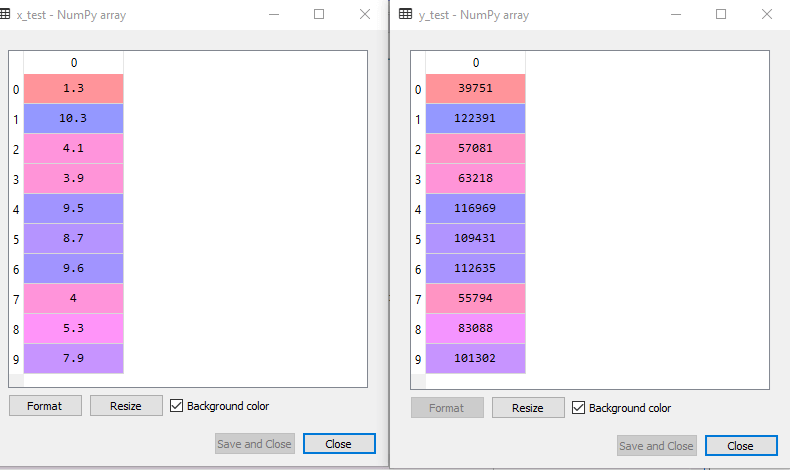


In the above output image, we can see the X (independent) variable and Y (dependent) variable has been extracted from the given dataset.

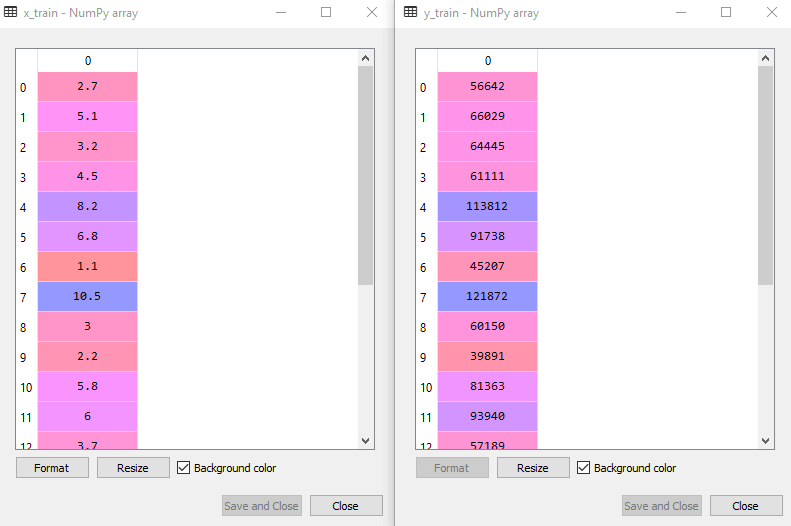
* Next, we will split both variables into the test set and training set. We have 30 observations, so we will take 20 observations for the training set and 10 observations for the test set. We are splitting our dataset so that we can train our model using a training dataset and then test the model using a test dataset. The code for this is given below:
* # Splitting the dataset into training and test set.
* from sklearn.model\_selection **import** train\_test\_split
* x\_train, x\_test, y\_train, y\_test= train\_test\_split(x, y, test\_size= 1/3, random\_state=0)

By executing the above code, we will get x-test, x-train and y-test, y-train dataset. Consider the below images:

Test-dataset:



Training Dataset:



* For simple linear Regression, we will not use Feature Scaling. Because Python libraries take care of it for some cases, so we don't need to perform it here. Now, our dataset is well prepared to work on it and we are going to start building a Simple Linear Regression model for the given problem.

Step-2: Fitting the Simple Linear Regression to the Training Set:

Now the second step is to fit our model to the training dataset. To do so, we will import the LinearRegression class of the linear\_model library from the scikit learn. After importing the class, we are going to create an object of the class named as a regressor. The code for this is given below:

* #Fitting the Simple Linear Regression model to the training dataset
* from sklearn.linear\_model **import** LinearRegression
* regressor= LinearRegression()
* regressor.fit(x\_train, y\_train)

In the above code, we have used a fit() method to fit our Simple Linear Regression object to the training set. In the fit() function, we have passed the x\_train and y\_train, which is our training dataset for the dependent and an independent variable. We have fitted our regressor object to the training set so that the model can easily learn the correlations between the predictor and target variables. After executing the above lines of code, we will get the below output.

Output:

*Out[7]: LinearRegression(copy\_X=True, fit\_intercept=True, n\_jobs=None, normalize=False)*

Step: 3. Prediction of test set result:

dependent (salary) and an independent variable (Experience). So, now, our model is ready to predict the output for the new observations. In this step, we will provide the test dataset (new observations) to the model to check whether it can predict the correct output or not.

We will create a prediction vector y\_pred, and x\_pred, which will contain predictions of test dataset, and prediction of training set respectively.

* #Prediction of Test and Training set result
* y\_pred= regressor.predict(x\_test)
* x\_pred= regressor.predict(x\_train)

On executing the above lines of code, two variables named y\_pred and x\_pred will generate in the variable explorer options that contain salary predictions for the training set and test set.

Output:

You can check the variable by clicking on the variable explorer option in the IDE, and also compare the result by comparing values from y\_pred and y\_test. By comparing these values, we can check how good our model is performing.

Step: 4. visualizing the Training set results:

Now in this step, we will visualize the training set result. To do so, we will use the scatter() function of the pyplot library, which we have already imported in the pre-processing step. The scatter () function will create a scatter plot of observations.

In the x-axis, we will plot the Years of Experience of employees and on the y-axis, salary of employees. In the function, we will pass the real values of training set, which means a year of experience x\_train, training set of Salaries y\_train, and color of the observations. Here we are taking a green color for the observation, but it can be any color as per the choice.

Now, we need to plot the regression line, so for this, we will use the plot() function of the pyplot library. In this function, we will pass the years of experience for training set, predicted salary for training set x\_pred, and color of the line.

Next, we will give the title for the plot. So here, we will use the title() function of the pyplot library and pass the name ("Salary vs Experience (Training Dataset)".

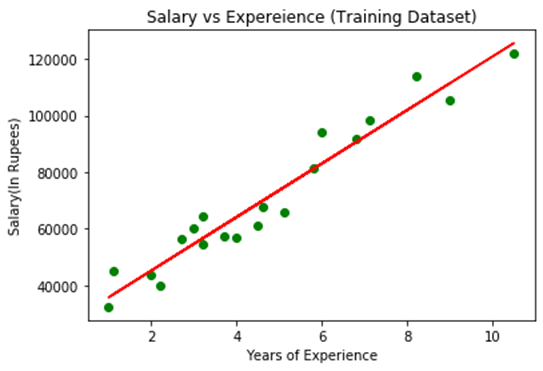
After that, we will assign labels for x-axis and y-axis using xlabel() and ylabel() function.

Finally, we will represent all above things in a graph using show(). The code is given below:

* mtp.scatter(x\_train, y\_train, color="green")
* mtp.plot(x\_train, x\_pred, color="red")
* mtp.title("Salary vs Experience (Training Dataset)")
* mtp.xlabel("Years of Experience")
* mtp.ylabel("Salary(In Rupees)")
* mtp.show()

Output:

By executing the above lines of code, we will get the below graph plot as an output.



In the above plot, we can see the real values observations in green dots and predicted values are covered by the red regression line. The regression line shows a correlation between the dependent and independent variable.

The good fit of the line can be observed by calculating the difference between actual values and predicted values. But as we can see in the above plot, most of the observations are close to the regression line, hence our model is good for the training set.

Step: 5. visualizing the Test set results:

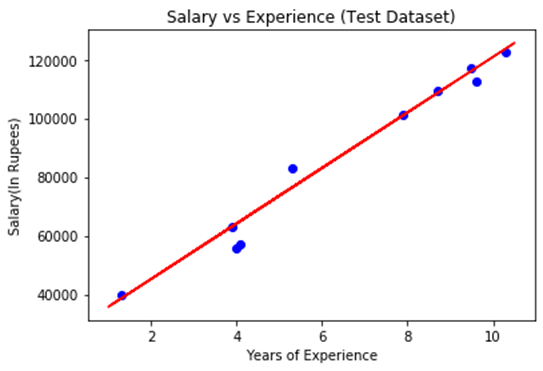
In the previous step, we have visualized the performance of our model on the training set. Now, we will do the same for the Test set. The complete code will remain the same as the above code, except in this, we will use x\_test, and y\_test instead of x\_train and y\_train.

Here we are also changing the color of observations and regression line to differentiate between the two plots, but it is optional.

* #visualizing the Test set results
* mtp.scatter(x\_test, y\_test, color="blue")
* mtp.plot(x\_train, x\_pred, color="red")
* mtp.title("Salary vs Experience (Test Dataset)")
* mtp.xlabel("Years of Experience")
* mtp.ylabel("Salary(In Rupees)")
* mtp.show()

Output:

By executing the above line of code, we will get the output as:

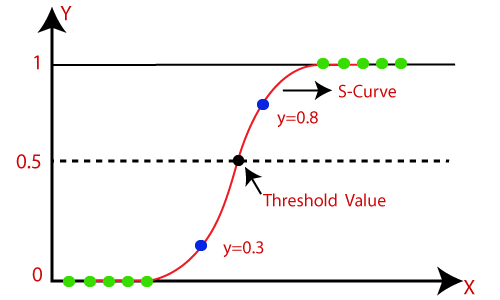


In the above plot, there are observations given by the blue color, and prediction is given by the red regression line. As we can see, most of the observations are close to the regression line, hence we can say our Simple Linear Regression is a good model and able to make good predictions.

**Practical no 2:**

# Logistic Regression in Machine Learning

* Logistic regression is one of the most popular Machine Learning algorithms, which comes under the Supervised Learning technique. It is used for predicting the categorical dependent variable using a given set of independent variables.
* Logistic regression predicts the output of a categorical dependent variable. Therefore the outcome must be a categorical or discrete value. It can be either Yes or No, 0 or 1, true or False, etc. but instead of giving the exact value as 0 and 1, it gives the probabilistic values which lie between 0 and 1.
* Logistic Regression is much similar to Linear Regression except that how they are used. Linear Regression is used for solving Regression problems, whereas Logistic regression is used for solving the classification problems.
* In Logistic regression, instead of fitting a regression line, we fit an "S" shaped logistic function, which predicts two maximum values (0 or 1).
* The curve from the logistic function indicates the likelihood of something such as whether the cells are cancerous or not, a mouse is obese or not based on its weight, etc.
* Logistic Regression is a significant machine learning algorithm because it has the ability to provide probabilities and classify new data using continuous and discrete datasets.
* Logistic Regression can be used to classify the observations using different types of data and can easily determine the most effective variables used for the classification. The below image is showing the logistic function:



#### *Note: Logistic regression uses the concept of predictive modeling as regression; therefore, it is called logistic regression, but is used to classify samples; Therefore, it falls under the classification algorithm.*

## Logistic Function (Sigmoid Function):

* The sigmoid function is a mathematical function used to map the predicted values to probabilities.
* It maps any real value into another value within a range of 0 and 1.
* The value of the logistic regression must be between 0 and 1, which cannot go beyond this limit, so it forms a curve like the "S" form. The S-form curve is called the Sigmoid function or the logistic function.
* In logistic regression, we use the concept of the threshold value, which defines the probability of either 0 or 1. Such as values above the threshold value tends to 1, and a value below the threshold values tends to 0.

## Assumptions for Logistic Regression:

* The dependent variable must be categorical in nature.
* The independent variable should not have multi-collinearity.

## Logistic Regression Equation:

The Logistic regression equation can be obtained from the Linear Regression equation. The mathematical steps to get Logistic Regression equations are given below:

* We know the equation of the straight line can be written as:

Logistic Regression in Machine Learning

* In Logistic Regression y can be between 0 and 1 only, so for this let's divide the above equation by (1-y):

Logistic Regression in Machine Learning

* But we need range between -[infinity] to +[infinity], then take logarithm of the equation it will become:

Logistic Regression in Machine Learning

The above equation is the final equation for Logistic Regression.

## Type of Logistic Regression:

On the basis of the categories, Logistic Regression can be classified into three types:

* Binomial: In binomial Logistic regression, there can be only two possible types of the dependent variables, such as 0 or 1, Pass or Fail, etc.
* Multinomial: In multinomial Logistic regression, there can be 3 or more possible unordered types of the dependent variable, such as "cat", "dogs", or "sheep"
* Ordinal: In ordinal Logistic regression, there can be 3 or more possible ordered types of dependent variables, such as "low", "Medium", or "High".

## Python Implementation of Logistic Regression (Binomial)

To understand the implementation of Logistic Regression in Python, we will use the below example:

Example: There is a dataset given which contains the information of various users obtained from the social networking sites. There is a car making company that has recently launched a new SUV car. So the company wanted to check how many users from the dataset, wants to purchase the car.

For this problem, we will build a Machine Learning model using the Logistic regression algorithm. The dataset is shown in the below image. In this problem, we will predict the purchased variable (Dependent Variable) by using age and salary (Independent variables).

Steps in Logistic Regression: To implement the Logistic Regression using Python, we will use the same steps as we have done in previous topics of Regression. Below are the steps:

* Data Pre-processing step
* Fitting Logistic Regression to the Training set
* Predicting the test result
* Test accuracy of the result(Creation of Confusion matrix)
* Visualizing the test set result.

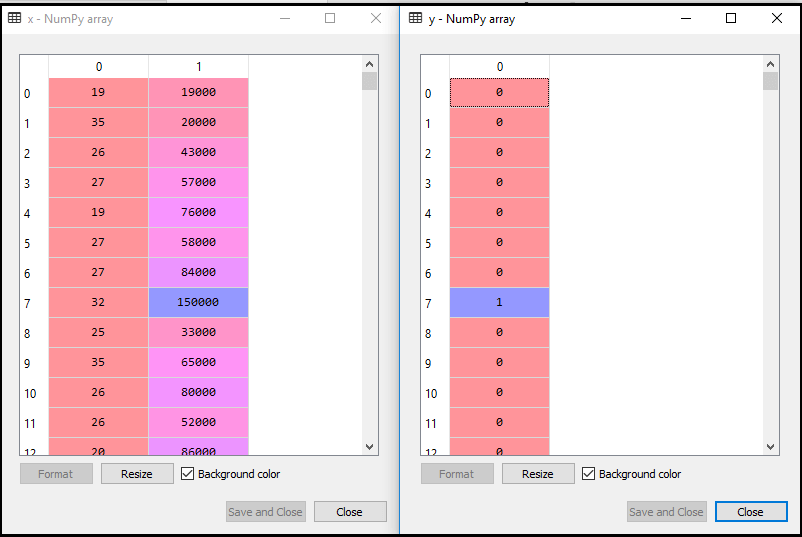
1. Data Pre-processing step: In this step, we will pre-process/prepare the data so that we can use it in our code efficiently. It will be the same as we have done in Data pre-processing topic. The code for this is given below:

* #Data Pre-procesing Step
* # importing libraries
* **import** numpy as nm
* **import** matplotlib.pyplot as mtp
* **import** pandas as pd
* #importing datasets
* data\_set= pd.read\_csv('user\_data.csv')

By executing the above lines of code, we will get the dataset as the output. Consider the given image:

* #Extracting Independent and dependent Variable
* x= data\_set.iloc[:, [2,3]].values
* y= data\_set.iloc[:, 4].values

In the above code, we have taken [2, 3] for x because our independent variables are age and salary, which are at index 2, 3. And we have taken 4 for y variable because our dependent variable is at index 4. The output will be:



Now we will split the dataset into a training set and test set. Below is the code for it:

* # Splitting the dataset into training and test set.
* from sklearn.model\_selection **import** train\_test\_split
* x\_train, x\_test, y\_train, y\_test= train\_test\_split(x, y, test\_size= 0.25, random\_state=0)

The output for this is given below:



2. Fitting Logistic Regression to the Training set:

We have well prepared our dataset, and now we will train the dataset using the training set. For providing training or fitting the model to the training set, we will import the LogisticRegression class of the sklearn library.

After importing the class, we will create a classifier object and use it to fit the model to the logistic regression. Below is the code for it:

* #Fitting Logistic Regression to the training set
* from sklearn.linear\_model **import** LogisticRegression
* classifier= LogisticRegression(random\_state=0)
* classifier.fit(x\_train, y\_train)

Output: By executing the above code, we will get the below output:

Out[5]:

* LogisticRegression(C=1.0, class\_weight=None, dual=False, fit\_intercept=True,
* intercept\_scaling=1, l1\_ratio=None, max\_iter=100,
* multi\_class='warn', n\_jobs=None, penalty='l2',
* random\_state=0, solver='warn', tol=0.0001, verbose=0,
* warm\_start=False)

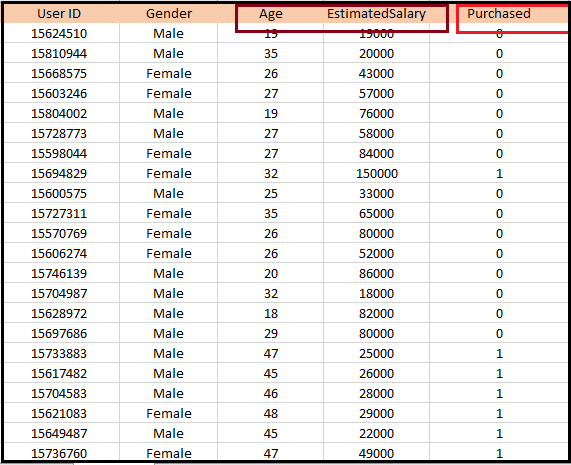
Hence our model is well fitted to the training set.

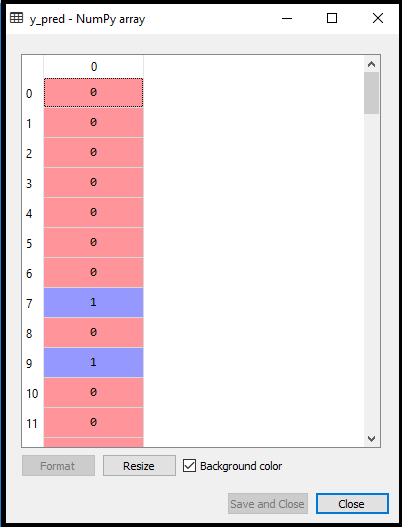
3. Predicting the Test Result

Our model is well trained on the training set, so we will now predict the result by using test set data. Below is the code for it:

* #Predicting the test set result
* y\_pred= classifier.predict(x\_test)

In the above code, we have created a y\_pred vector to predict the test set result.

Output: By executing the above code, a new vector (y\_pred) will be created under the variable explorer option. It can be seen as:



The above output image shows the corresponding predicted users who want to purchase or not purchase the car.

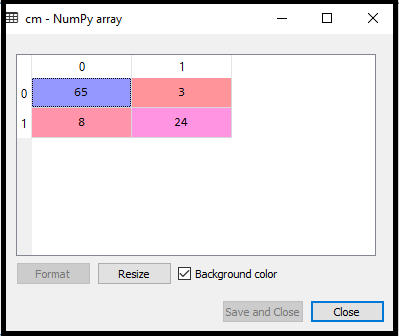
4. Test Accuracy of the result

Now we will create the confusion matrix here to check the accuracy of the classification. To create it, we need to import the confusion\_matrix function of the sklearn library. After importing the function, we will call it using a new variable cm. The function takes two parameters, mainly y\_true( the actual values) and y\_pred (the targeted value return by the classifier). Below is the code for it:

* # Creating the Confusion Matrix
* from sklearn.metrics import confusion\_matrix
* cm = confusion\_matrix(y\_test, y\_pred)

Output:

By executing the above code, a new confusion matrix will be created. Consider the below image:



We can find the accuracy of the predicted result by interpreting the confusion matrix. By above output, we can interpret that 65+24= 89 (Correct Output) and 8+3= 11(Incorrect Output).

5. Visualizing the training set result

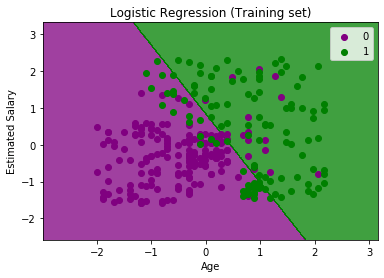
Finally, we will visualize the training set result. To visualize the result, we will use ListedColormap class of matplotlib library. Below is the code for it:

* #Visualizing the training set result
* from matplotlib.colors **import** ListedColormap
* x\_set, y\_set = x\_train, y\_train
* x1, x2 = nm.meshgrid(nm.arange(start = x\_set[:, 0].min() - 1, stop = x\_set[:, 0].max() + 1, step =0.01),
* nm.arange(start = x\_set[:, 1].min() - 1, stop = x\_set[:, 1].max() + 1, step = 0.01))
* mtp.contourf(x1, x2, classifier.predict(nm.array([x1.ravel(), x2.ravel()]).T).reshape(x1.shape),
* alpha = 0.75, cmap = ListedColormap(('purple','green' )))
* mtp.xlim(x1.min(), x1.max())
* mtp.ylim(x2.min(), x2.max())
* **for** i, j in enumerate(nm.unique(y\_set)):
* mtp.scatter(x\_set[y\_set == j, 0], x\_set[y\_set == j, 1],
* c = ListedColormap(('purple', 'green'))(i), label = j)
* mtp.title('Logistic Regression (Training set)')
* mtp.xlabel('Age')
* mtp.ylabel('Estimated Salary')
* mtp.legend()
* mtp.show()

In the above code, we have imported the ListedColormap class of Matplotlib library to create the colormap for visualizing the result. We have created two new variables x\_set and y\_set to replace x\_train and y\_train. After that, we have used the nm.meshgrid command to create a rectangular grid, which has a range of -1(minimum) to 1 (maximum). The pixel points we have taken are of 0.01 resolution.

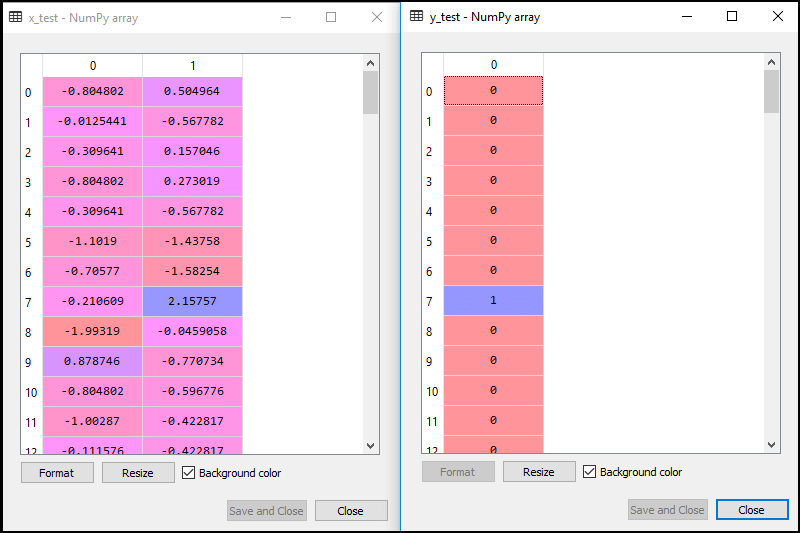
To create a filled contour, we have used mtp.contourf command, it will create regions of provided colors (purple and green). In this function, we have passed the classifier.predict to show the predicted data points predicted by the classifier.

Output: By executing the above code, we will get the below output:

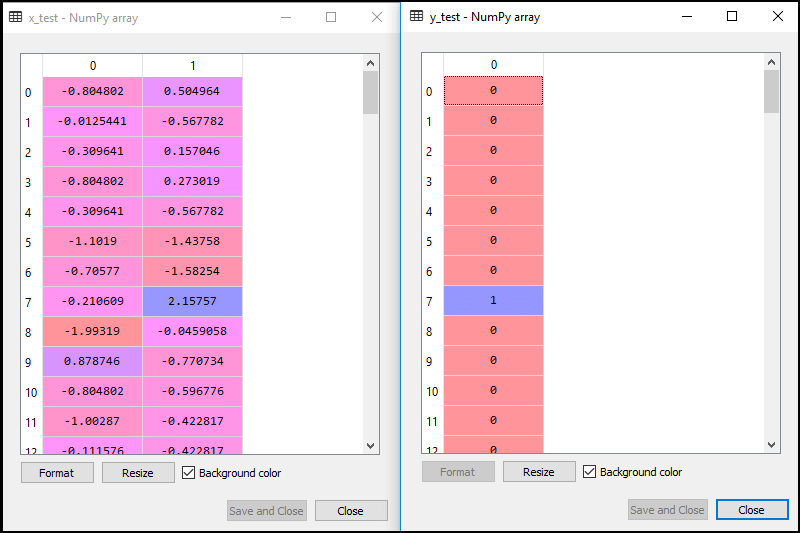


The graph can be explained in the below points:

* In the above graph, we can see that there are some Green points within the green region and Purple points within the purple region.
* All these data points are the observation points from the training set, which shows the result for purchased variables.
* This graph is made by using two independent variables i.e., Age on the x-axis and Estimated salary on the y-axis.
* The purple point observations are for which purchased (dependent variable) is probably 0, i.e., users who did not purchase the SUV car.
* The green point observations are for which purchased (dependent variable) is probably 1 means user who purchased the SUV car.
* We can also estimate from the graph that the users who are younger with low salary, did not purchase the car, whereas older users with high estimated salary purchased the car.
* But there are some purple points in the green region (Buying the car) and some green points in the purple region(Not buying the car). So we can say that younger users with a high estimated salary purchased the car, whereas an older user with a low estimated salary did not purchase the car.

For test set: 

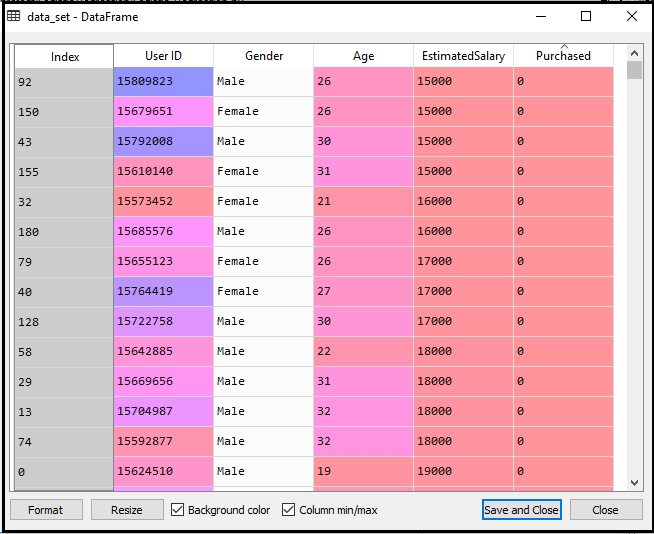
For training set:



In logistic regression, we will do feature scaling because we want accurate result of predictions. Here we will only scale the independent variable because dependent variable have only 0 and 1 values. Below is the code for it:

* #feature Scaling
* from sklearn.preprocessing **import** StandardScaler
* st\_x= StandardScaler()
* x\_train= st\_x.fit\_transform(x\_train)
* x\_test= st\_x.transform(x\_test)

The scaled output is given below:



Now, we will extract the dependent and independent variables from the given dataset. Below is the code for it:

**Practical no : 6**

To train an SVM (Support Vector Machine) regressor on the California Housing Dataset, we can use Python with libraries like scikit-learn and pandas. Below is an example code that shows how to load the dataset, preprocess it, and train an SVM regressor.

### **Step-by-Step :**

1. **Install Required Libraries:** Make sure you have the necessary libraries installed. You can install them via pip if they are not installed:

**pip install scikit-learn pandas matplotlib**

2. Code to Train an SVM Regressor:

**# Import necessary libraries**

**import numpy as np**

**import pandas as pd**

**from sklearn.datasets import fetch\_california\_housing**

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

**from sklearn.preprocessing import StandardScaler**

**from sklearn.svm import SVR**

**from sklearn.metrics import mean\_squared\_error**

**import matplotlib.pyplot as plt**

**# Load the California Housing Dataset**

**california\_housing = fetch\_california\_housing()**

**X = california\_housing.data**

**y = california\_housing.target**

**# Split the dataset into training and testing sets**

**X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)**

**# Feature scaling (important for SVM)**

**scaler = StandardScaler()**

**X\_train\_scaled = scaler.fit\_transform(X\_train)**

**X\_test\_scaled = scaler.transform(X\_test)**

**# Train the SVM regressor**

**svm\_regressor = SVR(kernel='rbf') # You can experiment with different kernels like 'linear', 'poly', etc.**

**svm\_regressor.fit(X\_train\_scaled, y\_train)**

**# Predict on the test set**

**y\_pred = svm\_regressor.predict(X\_test\_scaled)**

**# Evaluate the model**

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

**rmse = np.sqrt(mse)**

**print(f"Root Mean Squared Error (RMSE): {rmse}")**

**# Plot the true vs predicted values**

**plt.scatter(y\_test, y\_pred, alpha=0.5)**

**plt.plot([min(y\_test), max(y\_test)], [min(y\_test), max(y\_test)], color='red', lw=2)**

**plt.xlabel('True Values')**

**plt.ylabel('Predicted Values')**

**plt.title('True vs Predicted Values (SVM Regressor)')**

**plt.show()**

**Data Loading:** We use fetch\_california\_housing() to load the California Housing dataset. This dataset consists of various features like the average income, latitude, longitude, etc., to predict the median house value.

**Data Splitting:** We split the data into training and testing sets using train\_test\_split (80% training, 20% testing).

**Feature Scaling:** SVMs require feature scaling to work effectively, so we use StandardScaler to normalize the data.

**Model Training:** We use the SVR (Support Vector Regression) model with a Radial Basis Function (rbf) kernel. You can try other kernels as well, such as 'linear' or 'poly'.

**Model Evaluation:** We calculate the Root Mean Squared Error (RMSE) to evaluate the performance of the model.

**Plotting:** We create a scatter plot of the true vs predicted values to visually inspect the model's performance.

### **Output:**

**Root Mean Squared Error (RMSE)**:  
The model's performance will be evaluated using RMSE, which is a common metric for regression tasks. The exact value of RMSE will depend on various factors like the kernel used and the hyperparameters of the model. However, you can expect the RMSE to be in the range of around 0.5 to 1.5 for this dataset.  
Example output might look like this:  
  
Root Mean Squared Error (RMSE): 0.756

1. **Plot (True vs Predicted Values)**:  
   The scatter plot will show the true values (actual house prices) on the x-axis and the predicted values (values predicted by the SVM) on the y-axis. Ideally, the points should be close to the red diagonal line, which represents perfect predictions where the true value equals the predicted value.  
   The plot would look like this:
   * Points distributed around a red diagonal line.
   * A well-performing model should have points close to the line.
2. **Interpretation**:
   * If the points are scattered significantly away from the line, it indicates that the model is not performing well.
   * If most points are near the line, the model has good predictive accuracy.

### **Improving Performance:**

* **Hyperparameter Tuning:**You can try tuning the C, gamma, and epsilon parameters of the SVM regressor to improve the model's performance.
  + You can use grid search or random search (GridSearchCV or RandomizedSearchCV from scikit-learn) to find the optimal set of hyperparameters.
* **Cross-validation:**Instead of a single train-test split, using cross-validation (e.g., cross\_val\_score) could help provide more robust estimates of model performance.

**Practical no 8**

Implement MLP for classification of handwritten digits (MNIST Dataset)

Here’s a step-by-step implementation of a **Multilayer Perceptron (MLP)** using TensorFlow/Keras for the classification of **handwritten digits** from the MNIST dataset.

### **Steps:**

1. **Load the MNIST Dataset**
2. **Preprocess the Data**
3. **Build the MLP Model**
4. **Compile the Model**
5. **Train the Model**
6. **Evaluate the Model**

### **1. Import Libraries**

import tensorflow as tf

from tensorflow.keras import layers, models

from tensorflow.keras.datasets import mnist

from tensorflow.keras.utils import to\_categorical

### **2. Load and Preprocess the MNIST Dataset**

The MNIST dataset contains 28x28 grayscale images of handwritten digits (0 to 9). We will normalize the pixel values to the range [0, 1] and one-hot encode the labels.

# Load the dataset

(x\_train, y\_train), (x\_test, y\_test) = mnist.load\_data()

# Normalize the pixel values to be between 0 and 1

x\_train, x\_test = x\_train / 255.0, x\_test / 255.0

# Flatten the images into vectors of size 784 (28 \* 28)

x\_train = x\_train.reshape(-1, 28 \* 28)

x\_test = x\_test.reshape(-1, 28 \* 28)

# One-hot encode the labels

y\_train = to\_categorical(y\_train, 10)

y\_test = to\_categorical(y\_test, 10)

### **3. Build the MLP Model**

We will create a simple MLP with two hidden layers and an output layer with 10 units (for the 10 digit classes). The activation function for the hidden layers will be **ReLU**, and the output layer will use **softmax** for multi-class classification.

# Build the MLP model

model = models.Sequential([

layers.InputLayer(input\_shape=(28 \* 28,)), # Input layer (flattened 28x28 images)

layers.Dense(128, activation='relu'), # First hidden layer with 128 neurons

layers.Dense(64, activation='relu'), # Second hidden layer with 64 neurons

layers.Dense(10, activation='softmax') # Output layer with 10 classes (digits 0-9)

])

### **4. Compile the Model**

We will use **categorical cross-entropy** as the loss function, which is standard for multi-class classification, and the **Adam optimizer** for optimization.

# Compile the model

model.compile(

optimizer='adam', # Adam optimizer

loss='categorical\_crossentropy', # Categorical cross-entropy loss

metrics=['accuracy'] # Track accuracy during training

)

### **5. Train the Model**

We will train the model for 10 epochs using the training data. The validation data will be the test set to evaluate the performance after each epoch.

# Train the model

model.fit(x\_train, y\_train, epochs=10, batch\_size=32, validation\_data=(x\_test, y\_test))

### **6. Evaluate the Model**

After training, we can evaluate the model’s performance on the test data.

# Evaluate the model on the test set

test\_loss, test\_acc = model.evaluate(x\_test, y\_test)

print(f"Test Loss: {test\_loss}")

print(f"Test Accuracy: {test\_acc}")

### **Full Code:**

import tensorflow as tf

from tensorflow.keras import layers, models

from tensorflow.keras.datasets import mnist

from tensorflow.keras.utils import to\_categorical

# 1. Load and preprocess the dataset

(x\_train, y\_train), (x\_test, y\_test) = mnist.load\_data()

# Normalize the pixel values to be between 0 and 1

x\_train, x\_test = x\_train / 255.0, x\_test / 255.0

# Flatten the images into vectors of size 784 (28 \* 28)

x\_train = x\_train.reshape(-1, 28 \* 28)

x\_test = x\_test.reshape(-1, 28 \* 28)

# One-hot encode the labels

y\_train = to\_categorical(y\_train, 10)

y\_test = to\_categorical(y\_test, 10)

# 2. Build the MLP model

model = models.Sequential([

layers.InputLayer(input\_shape=(28 \* 28,)), # Input layer (flattened 28x28 images)

layers.Dense(128, activation='relu'), # First hidden layer with 128 neurons

layers.Dense(64, activation='relu'), # Second hidden layer with 64 neurons

layers.Dense(10, activation='softmax') # Output layer with 10 classes (digits 0-9)

])

# 3. Compile the model

model.compile(

optimizer='adam', # Adam optimizer

loss='categorical\_crossentropy', # Categorical cross-entropy loss

metrics=['accuracy'] # Track accuracy during training

)

# 4. Train the model

model.fit(x\_train, y\_train, epochs=10, batch\_size=32, validation\_data=(x\_test, y\_test))

# 5. Evaluate the model

test\_loss, test\_acc = model.evaluate(x\_test, y\_test)

print(f"Test Loss: {test\_loss}")

print(f"Test Accuracy: {test\_acc}")

### **Explanation:**

1. **Data Preprocessing**:
   * **Normalization**: The pixel values of the images range from 0 to 255. We normalize them to a range between 0 and 1 for better performance during training.
   * **Reshaping**: The original MNIST images are 28x28 pixels. We flatten them into vectors of size 784 (28 \* 28).
   * **One-hot Encoding**: The labels (digits 0-9) are one-hot encoded, converting each label to a vector of size 10, with a 1 at the index corresponding to the digit and 0s elsewhere.
2. **Model Architecture**:
   * **Input Layer**: The input consists of a vector of 784 values (flattened 28x28 image).
   * **Hidden Layers**: We use two **Dense** layers with **ReLU** activation. The first layer has 128 neurons, and the second has 64 neurons.
   * **Output Layer**: The output layer consists of 10 neurons (for 10 classes, digits 0-9) with **softmax** activation, which is used for multi-class classification.
3. **Compilation and Training**:
   * We use **Adam optimizer** and **categorical cross-entropy** as the loss function for multi-class classification.
   * We train the model for 10 epochs with a batch size of 32. We also use the test set as validation data to monitor the model’s performance during training.
4. **Evaluation**:
   * After training, we evaluate the model on the test set to check its accuracy.

**Practical No 9**

Classification of images of clothing using Tensorflow (Fashion MNIST dataset)

Here’s how you can classify images of clothing using TensorFlow with the **Fashion MNIST dataset**, which contains 60,000 28x28 grayscale images of 10 different types of clothing (such as t-shirts, trousers, etc.). We'll implement this using a **Multilayer Perceptron (MLP)** model to perform classification.

### **Steps:**

1. **Load the Fashion MNIST Dataset**
2. **Preprocess the Data**
3. **Build the Model**
4. **Compile the Model**
5. **Train the Model**
6. **Evaluate the Model**

### **1. Import Libraries**

import tensorflow as tf

from tensorflow.keras import layers, models

from tensorflow.keras.datasets import fashion\_mnist

from tensorflow.keras.utils import to\_categorical

### **2. Load and Preprocess the Fashion MNIST Dataset**

Fashion MNIST images are 28x28 pixels, and we'll normalize the images to the range [0, 1] and one-hot encode the labels.

# Load the Fashion MNIST dataset

(x\_train, y\_train), (x\_test, y\_test) = fashion\_mnist.load\_data()

# Normalize the images to be between 0 and 1

x\_train, x\_test = x\_train / 255.0, x\_test / 255.0

# Flatten the images to vectors of size 784 (28 \* 28)

x\_train = x\_train.reshape(-1, 28 \* 28)

x\_test = x\_test.reshape(-1, 28 \* 28)

# One-hot encode the labels

y\_train = to\_categorical(y\_train, 10)

y\_test = to\_categorical(y\_test, 10)

### **3. Build the Model**

We’ll build a **Multilayer Perceptron (MLP)** model with one input layer, two hidden layers, and an output layer. The hidden layers will use **ReLU** activation, and the output layer will use **softmax** for multi-class classification.

# Build the MLP model

model = models.Sequential([

layers.InputLayer(input\_shape=(28 \* 28,)), # Input layer (flattened 28x28 images)

layers.Dense(128, activation='relu'), # First hidden layer with 128 neurons

layers.Dense(64, activation='relu'), # Second hidden layer with 64 neurons

layers.Dense(10, activation='softmax') # Output layer with 10 classes (clothing types)

])

### **4. Compile the Model**

We’ll use **categorical cross-entropy** as the loss function, **Adam optimizer** for optimization, and **accuracy** as the evaluation metric.

# Compile the model

model.compile(

optimizer='adam', # Adam optimizer

loss='categorical\_crossentropy', # Categorical cross-entropy loss

metrics=['accuracy'] # Track accuracy during training

)

### **5. Train the Model**

We’ll train the model using the training data. The validation data will be the test set to monitor performance during training.

# Train the model

model.fit(x\_train, y\_train, epochs=10, batch\_size=32, validation\_data=(x\_test, y\_test))

### **6. Evaluate the Model**

After training, we can evaluate the model on the test set to see how well it performs on unseen data.

# Evaluate the model on the test set

test\_loss, test\_acc = model.evaluate(x\_test, y\_test)

print(f"Test Loss: {test\_loss}")

print(f"Test Accuracy: {test\_acc}")

### **Full Code:**

import tensorflow as tf

from tensorflow.keras import layers, models

from tensorflow.keras.datasets import fashion\_mnist

from tensorflow.keras.utils import to\_categorical

# 1. Load and preprocess the dataset

(x\_train, y\_train), (x\_test, y\_test) = fashion\_mnist.load\_data()

# Normalize the images to be between 0 and 1

x\_train, x\_test = x\_train / 255.0, x\_test / 255.0

# Flatten the images to vectors of size 784 (28 \* 28)

x\_train = x\_train.reshape(-1, 28 \* 28)

x\_test = x\_test.reshape(-1, 28 \* 28)

# One-hot encode the labels

y\_train = to\_categorical(y\_train, 10)

y\_test = to\_categorical(y\_test, 10)

# 2. Build the MLP model

model = models.Sequential([

layers.InputLayer(input\_shape=(28 \* 28,)), # Input layer (flattened 28x28 images)

layers.Dense(128, activation='relu'), # First hidden layer with 128 neurons

layers.Dense(64, activation='relu'), # Second hidden layer with 64 neurons

layers.Dense(10, activation='softmax') # Output layer with 10 classes (clothing types)

])

# 3. Compile the model

model.compile(

optimizer='adam', # Adam optimizer

loss='categorical\_crossentropy', # Categorical cross-entropy loss

metrics=['accuracy'] # Track accuracy during training

)

# 4. Train the model

model.fit(x\_train, y\_train, epochs=10, batch\_size=32, validation\_data=(x\_test, y\_test))

# 5. Evaluate the model

test\_loss, test\_acc = model.evaluate(x\_test, y\_test)

print(f"Test Loss: {test\_loss}")

print(f"Test Accuracy: {test\_acc}")

### **Explanation:**

1. **Data Preprocessing**:
   * **Normalization**: The pixel values of the images are between 0 and 255. We normalize them to the range [0, 1] by dividing by 255.
   * **Reshaping**: Fashion MNIST images are 28x28, and we flatten them into vectors of size 784 to use them in a fully connected MLP model.
   * **One-hot Encoding**: The labels (clothing types) are one-hot encoded, where each label is converted into a vector of length 10, with a 1 in the position corresponding to the correct clothing type.
2. **Model Architecture**:
   * **Input Layer**: The input layer has 784 neurons, corresponding to the 784 flattened pixel values of each image.
   * **Hidden Layers**: We have two hidden layers with 128 and 64 neurons, respectively. Each layer uses the **ReLU** activation function.
   * **Output Layer**: The output layer has 10 neurons (one for each clothing class) and uses the **softmax** activation function, which is appropriate for multi-class classification.
3. **Compilation and Training**:
   * We compile the model with the **Adam optimizer** and **categorical cross-entropy** loss function, which is suitable for multi-class classification problems.
   * The model is trained for 10 epochs with a batch size of 32. The validation data is the test set, so we can monitor the model’s performance during training.
4. **Evaluation**:
   * After training, the model is evaluated on the test set to determine its accuracy and loss on unseen data.

### **Example Output:**

After training, the model should output something like this:

plaintext

Copy

Epoch 1/10

1875/1875 [==============================] - 4s 2ms/step - loss: 0.5012 - accuracy: 0.8239 - val\_loss: 0.3686 - val\_accuracy: 0.8689

Epoch 2/10

1875/1875 [==============================] - 4s 2ms/step - loss: 0.3669 - accuracy: 0.8688 - val\_loss: 0.3450 - val\_accuracy: 0.8755

...

Test Loss: 0.3389497392177582

Test Accuracy: 0.8765000109672546

**Practical NO :8**

**Implement MLP for classification of handwritten digits (MNIST Dataset)**

an implementation of a Multi-Layer Perceptron (MLP) for classifying handwritten digits from the MNIST dataset using TensorFlow and Keras.

import tensorflow as tf

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Dense, Flatten

from tensorflow.keras.datasets import mnist

from tensorflow.keras.utils import to\_categorical

# Load and preprocess the MNIST dataset

(x\_train, y\_train), (x\_test, y\_test) = mnist.load\_data()

x\_train, x\_test = x\_train / 255.0, x\_test / 255.0 # Normalize pixel values

y\_train = to\_categorical(y\_train, 10)

y\_test = to\_categorical(y\_test, 10)

# Build the MLP model

model = Sequential([

Flatten(input\_shape=(28, 28)), # Flatten the 28x28 images into a vector

Dense(128, activation='relu'), # Hidden layer with ReLU activation

Dense(64, activation='relu'), # Another hidden layer

Dense(10, activation='softmax') # Output layer with softmax activation

])

# Compile the model

model.compile(optimizer='adam',

loss='categorical\_crossentropy',

metrics=['accuracy'])

# Train the model

model.fit(x\_train, y\_train, epochs=10, batch\_size=32, validation\_data=(x\_test, y\_test))

# Evaluate the model

test\_loss, test\_acc = model.evaluate(x\_test, y\_test, verbose=2)

print(f'\nTest accuracy: {test\_acc:.4f}')

This script:

* Loads and preprocesses the MNIST dataset (normalization and one-hot encoding).
* Defines a simple MLP model with two hidden layers.
* Compiles the model using the Adam optimizer and categorical cross-entropy loss.
* Trains the model for 10 epochs.
* Evaluates the model's performance on the test dataset.

### **Expected Output**

After running the script, you will see the following:

1. **Downloading the MNIST dataset** (if not already cached):

Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.npz

**Training Progress** (for 10 epochs):Epoch 1/10

1875/1875 [==============================] - 5s 2ms/step - loss: 0.2501 - accuracy: 0.9285 - val\_loss: 0.1234 - val\_accuracy: 0.9635

Epoch 2/10

1875/1875 [==============================] - 4s 2ms/step - loss: 0.1075 - accuracy: 0.9679 - val\_loss: 0.0894 - val\_accuracy: 0.9724

...

Epoch 10/10

1875/1875 [==============================] - 4s 2ms/step - loss: 0.0438 - accuracy: 0.9860 - val\_loss: 0.0732 - val\_accuracy: 0.9789

**Evaluation on Test Data**:

313/313 - 0s - loss: 0.0732 - accuracy: 0.9789

**Final Test Accuracy**:

Test accuracy: 0.9789

The model achieves **~97-98% accuracy** on the test dataset.

Training takes a few seconds per epoch.

The loss function decreases steadily, indicating good model learning.

**Practical no 7**

**Implement Batch Gradient Descent with early stopping for Softmax Regression.**

import numpy as np

import tensorflow as tf

from tensorflow.keras.datasets import mnist

from tensorflow.keras.utils import to\_categorical

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Dense, Flatten

from tensorflow.keras.callbacks import EarlyStopping

# Load and preprocess MNIST dataset

(x\_train, y\_train), (x\_test, y\_test) = mnist.load\_data()

x\_train, x\_test = x\_train / 255.0, x\_test / 255.0 # Normalize pixel values

y\_train, y\_test = to\_categorical(y\_train, 10), to\_categorical(y\_test, 10)

# Build Softmax Regression model

model = Sequential([

Flatten(input\_shape=(28, 28)),

Dense(10, activation='softmax') # Softmax regression

])

# Compile the model with SGD optimizer (Batch Gradient Descent)

model.compile(optimizer=tf.keras.optimizers.SGD(learning\_rate=0.1),

loss='categorical\_crossentropy',

metrics=['accuracy'])

# Implement Early Stopping

early\_stopping = EarlyStopping(monitor='val\_loss', patience=3, restore\_best\_weights=True)

# Train model using Batch Gradient Descent

model.fit(x\_train, y\_train, epochs=50, batch\_size=len(x\_train), validation\_data=(x\_test, y\_test), callbacks=[early\_stopping])

# Evaluate model

test\_loss, test\_acc = model.evaluate(x\_test, y\_test, verbose=2)

print(f'\nTest accuracy: {test\_acc:.4f}')

**Dataset:** MNIST (Handwritten digits)

**Model:** Softmax Regression

**Optimizer:** Stochastic Gradient Descent (Batch Gradient Descent)

**Early Stopping:** Stops training if validation loss does not improve for 3 consecutive epochs

**Batch Size:** Full dataset (Batch Gradient Descent)

**Practical no 10: Implement Regression to predict fuel efficiency using Tensorflow (Auto MPG dataset)**

import tensorflow as tf

import pandas as pd

import numpy as np

import matplotlib.pyplot as plt

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Dense, Normalization

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

from sklearn.preprocessing import StandardScaler

# Load dataset

dataset\_url = "http://archive.ics.uci.edu/ml/machine-learning-databases/auto-mpg/auto-mpg.data"

column\_names = ['MPG', 'Cylinders', 'Displacement', 'Horsepower', 'Weight', 'Acceleration', 'Model Year', 'Origin']

dataset = pd.read\_csv(dataset\_url, names=column\_names, na\_values="?", comment='\t', sep=" ", skipinitialspace=True)

# Drop missing values

dataset = dataset.dropna()

# Convert 'Origin' to one-hot encoding

dataset['Origin'] = dataset['Origin'].astype(int)

dataset = pd.get\_dummies(dataset, columns=['Origin'], prefix='', prefix\_sep='')

# Split features and labels

X = dataset.drop('MPG', axis=1)

y = dataset['MPG']

# Train-test split

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

# Normalize the data

scaler = StandardScaler()

X\_train = scaler.fit\_transform(X\_train)

X\_test = scaler.transform(X\_test)

# Build regression model

model = Sequential([

Dense(64, activation='relu', input\_shape=(X\_train.shape[1],)),

Dense(64, activation='relu'),

Dense(1) # Output layer for regression

])

# Compile the model

model.compile(optimizer=tf.keras.optimizers.Adam(learning\_rate=0.01), loss='mse', metrics=['mae'])

# Train the model

history = model.fit(X\_train, y\_train, epochs=100, batch\_size=32, validation\_data=(X\_test, y\_test), verbose=1)

# Evaluate model

loss, mae = model.evaluate(X\_test, y\_test, verbose=2)

print(f'\nTest Mean Absolute Error: {mae:.2f} MPG')

### **Overview of the Implementation**

* **Dataset:** Auto MPG dataset (downloaded from UCI)
* **Data Preprocessing:**
  + Handles missing values
  + Converts categorical 'Origin' column into one-hot encoding
  + Normalizes features using StandardScaler
* **Model:**
  + Fully connected feedforward neural network with two hidden layers (64 neurons each, ReLU activation)
* **Training:**
  + Optimized using Adam optimizer
  + Mean Squared Error (MSE) as the loss function
  + Monitors Mean Absolute Error (MAE)
* **Evaluation:**
  + Reports the MAE on the test dataset