

Simple+Linear+Regression+in+Python

October 2, 2024

1 Simple Linear Regression

In this notebook, we'll build a linear regression model to predict **Sales** using an appropriate predictor variable.

1.1 Step 1: Reading and Understanding the Data

Let's start with the following steps:

1. Importing data using the pandas library
2. Understanding the structure of the data

```
[3]: # Suppress Warnings
```

```
import warnings
warnings.filterwarnings('ignore')
```

```
[4]: # Import the numpy and pandas package
```

```
import numpy as np
import pandas as pd
```

```
[5]: # Read the given CSV file, and view some sample records
```

```
advertising = pd.read_csv("advertising.csv")
advertising.head()
```

```
[5]:
```

	TV	Radio	Newspaper	Sales
0	230.1	37.8	69.2	22.1
1	44.5	39.3	45.1	10.4
2	17.2	45.9	69.3	12.0
3	151.5	41.3	58.5	16.5
4	180.8	10.8	58.4	17.9

Let's inspect the various aspects of our dataframe

```
[7]: advertising.shape
```

```
[7]: (200, 4)
```

```
[8]: advertising.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 200 entries, 0 to 199
Data columns (total 4 columns):
 #   Column      Non-Null Count  Dtype
---  -
 0    TV         200 non-null    float64
 1   Radio       200 non-null    float64
 2  Newspaper   200 non-null    float64
 3    Sales      200 non-null    float64
dtypes: float64(4)
memory usage: 6.4 KB
```

```
[9]: advertising.describe()
```

```
[9]:
```

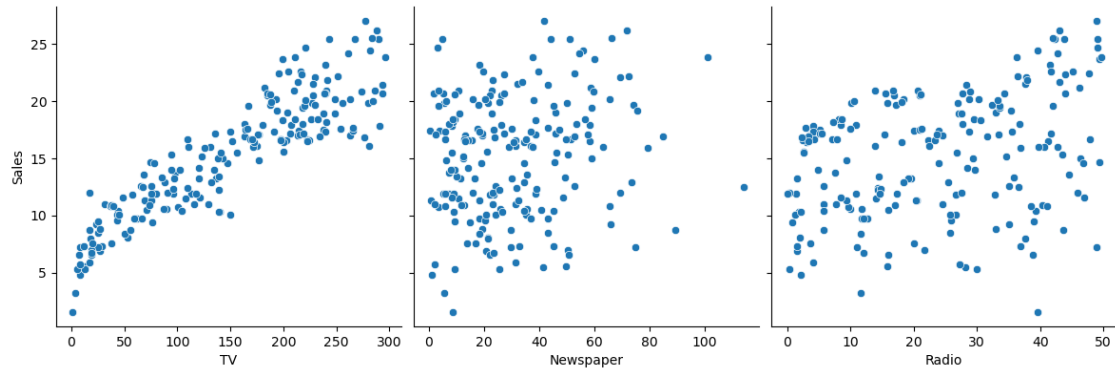
	TV	Radio	Newspaper	Sales
count	200.000000	200.000000	200.000000	200.000000
mean	147.042500	23.264000	30.554000	15.130500
std	85.854236	14.846809	21.778621	5.283892
min	0.700000	0.000000	0.300000	1.600000
25%	74.375000	9.975000	12.750000	11.000000
50%	149.750000	22.900000	25.750000	16.000000
75%	218.825000	36.525000	45.100000	19.050000
max	296.400000	49.600000	114.000000	27.000000

1.2 Step 2: Visualising the Data

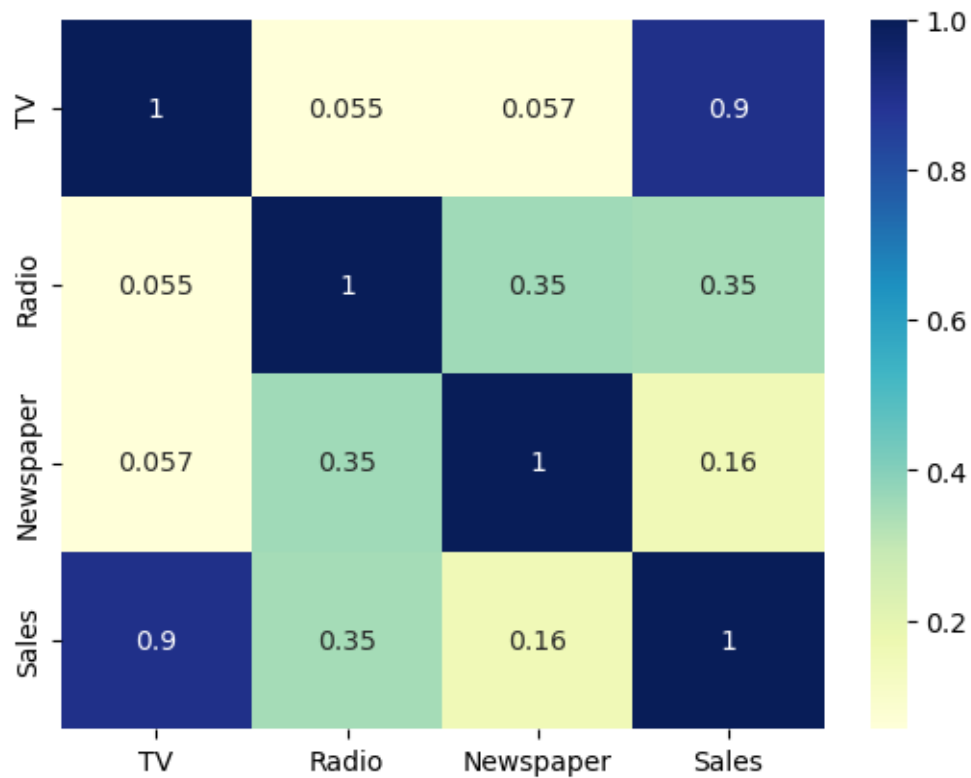
Let's now visualise our data using seaborn. We'll first make a pairplot of all the variables present to visualise which variables are most correlated to **Sales**.

```
[11]: import matplotlib.pyplot as plt
import seaborn as sns
```

```
[12]: sns.pairplot(advertising, x_vars=['TV', 'Newspaper', 'Radio'],
    ↪ y_vars='Sales',size=4, aspect=1, kind='scatter')
plt.show()
```



```
[13]: sns.heatmap(advertising.corr(), cmap="YlGnBu", annot = True)
plt.show()
```



As is visible from the pairplot and the heatmap, the variable TV seems to be most correlated with Sales. So let's go ahead and perform simple linear regression using TV as our feature variable.

1.2.1 Generic Steps in model building using statsmodels

We first assign the feature variable, TV, in this case, to the variable X and the response variable, Sales, to the variable y.

```
[17]: X = advertising['TV']  
      y = advertising['Sales']
```

Train-Test Split You now need to split our variable into training and testing sets. You'll perform this by importing `train_test_split` from the `sklearn.model_selection` library. It is usually a good practice to keep 70% of the data in your train dataset and the rest 30% in your test dataset

```
[19]: from sklearn.model_selection import train_test_split  
      X_train, X_test, y_train, y_test = train_test_split(X, y, train_size = 0.7,  
      ↪test_size = 0.3, random_state = 100)
```

```
[20]: # Let's now take a look at the train dataset  
  
      X_train.head()
```

```
[20]: 74      213.4  
      3       151.5  
      185     205.0  
      26      142.9  
      90      134.3  
      Name: TV, dtype: float64
```

```
[21]: y_train.head()
```

```
[21]: 74      17.0  
      3      16.5  
      185    22.6  
      26     15.0  
      90     14.0  
      Name: Sales, dtype: float64
```

Building a Linear Model You first need to import the `statsmodel.api` library using which you'll perform the linear regression.

```
[23]: import statsmodels.api as sm
```

By default, the `statsmodels` library fits a line on the dataset which passes through the origin. But in order to have an intercept, you need to manually use the `add_constant` attribute of `statsmodels`. And once you've added the constant to your `X_train` dataset, you can go ahead and fit a regression line using the OLS (Ordinary Least Squares) attribute of `statsmodels` as shown below

```
[25]: # Add a constant to get an intercept  
      X_train_sm = sm.add_constant(X_train)
```

```
# Fit the regression line using 'OLS'
lr = sm.OLS(y_train, X_train_sm).fit()
```

```
[26]: # Print the parameters, i.e. the intercept and the slope of the regression line
      ↪fitted
      lr.params
```

```
[26]: const    6.948683
      TV      0.054546
      dtype: float64
```

```
[27]: # Performing a summary operation lists out all the different parameters of the
      ↪regression line fitted
      print(lr.summary())
```

```

                                OLS Regression Results
=====
Dep. Variable:                  Sales    R-squared:                  0.816
Model:                            OLS    Adj. R-squared:              0.814
Method:                           Least Squares    F-statistic:                611.2
Date:                            Wed, 02 Oct 2024    Prob (F-statistic):         1.52e-52
Time:                            04:01:20    Log-Likelihood:             -321.12
No. Observations:                 140    AIC:                        646.2
Df Residuals:                     138    BIC:                        652.1
Df Model:                          1
Covariance Type:                  nonrobust
=====

```

	coef	std err	t	P> t	[0.025	0.975]
const	6.9487	0.385	18.068	0.000	6.188	7.709
TV	0.0545	0.002	24.722	0.000	0.050	0.059

```

=====
Omnibus:                        0.027    Durbin-Watson:              2.196
Prob(Omnibus):                  0.987    Jarque-Bera (JB):           0.150
Skew:                          -0.006    Prob(JB):                   0.928
Kurtosis:                       2.840    Cond. No.                    328.
=====

```

Notes:

[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.

Looking at some key statistics from the summary The values we are concerned with are -
 1. The coefficients and significance (p-values) 2. R-squared 3. F statistic and its significance

1. The coefficient for TV is 0.054, with a very low p value The coefficient is statistically significant. So the association is not purely by chance.

2. R - squared is 0.816 Meaning that 81.6% of the variance in Sales is explained by TV

This is a decent R-squared value.

3. F statistic has a very low p value (practically low)

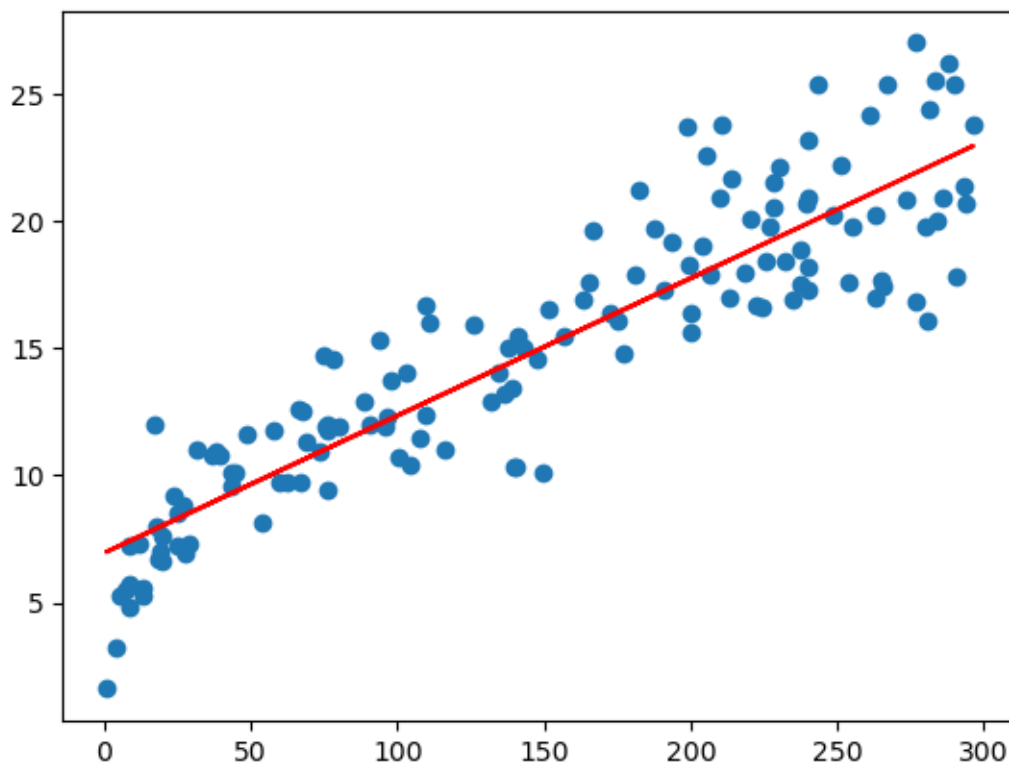
Meaning that the model fit is statistically significant, and the explained variance isn't purely by chance.

The fit is significant. Let's visualize how well the model fit the data.

From the parameters that we get, our linear regression equation becomes:

$$\text{\$ Sales} = 6.948 + 0.054 \times \text{TV} \text{\$}$$

```
[34]: plt.scatter(X_train, y_train)
plt.plot(X_train, 6.948 + 0.054*X_train, 'r')
plt.show()
```



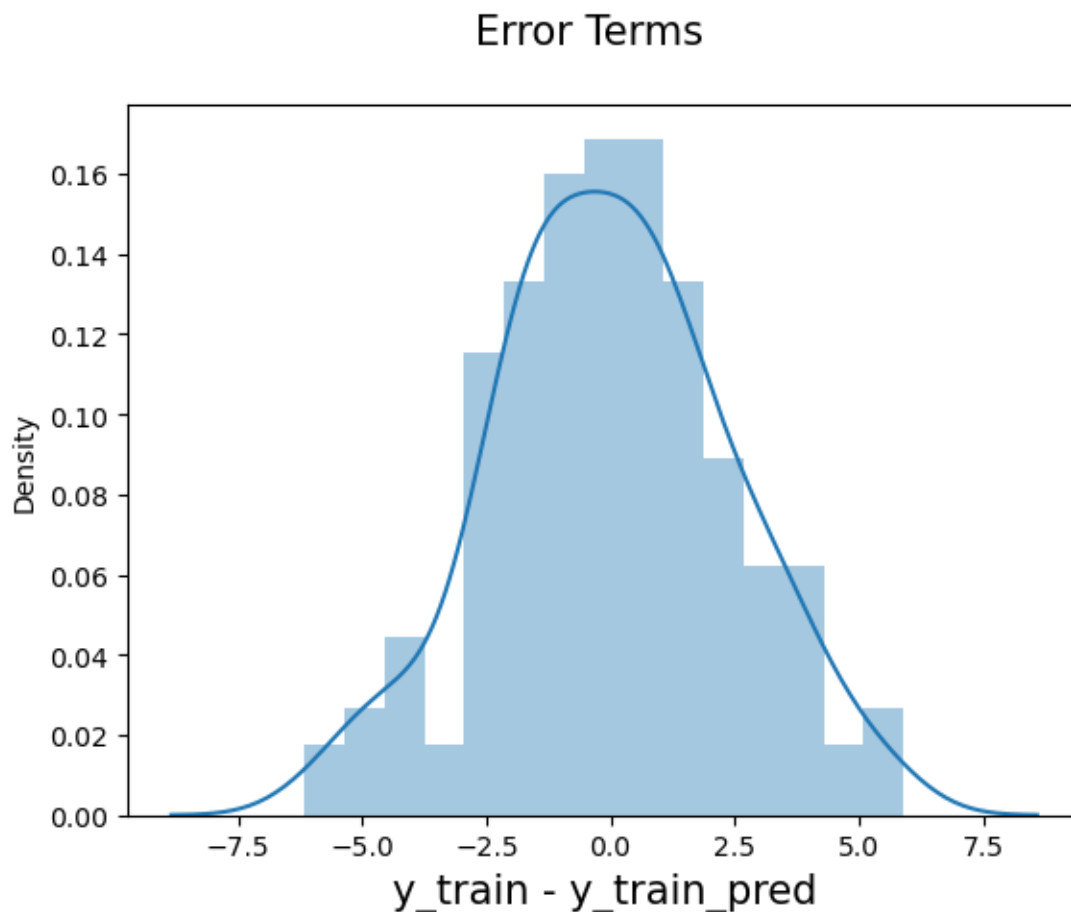
1.3 Step 4: Residual analysis

To validate assumptions of the model, and hence the reliability for inference

Distribution of the error terms We need to check if the error terms are also normally distributed (which is infact, one of the major assumptions of linear regression), let us plot the histogram of the error terms and see what it looks like.

```
[37]: y_train_pred = lr.predict(X_train_sm)
      res = (y_train - y_train_pred)

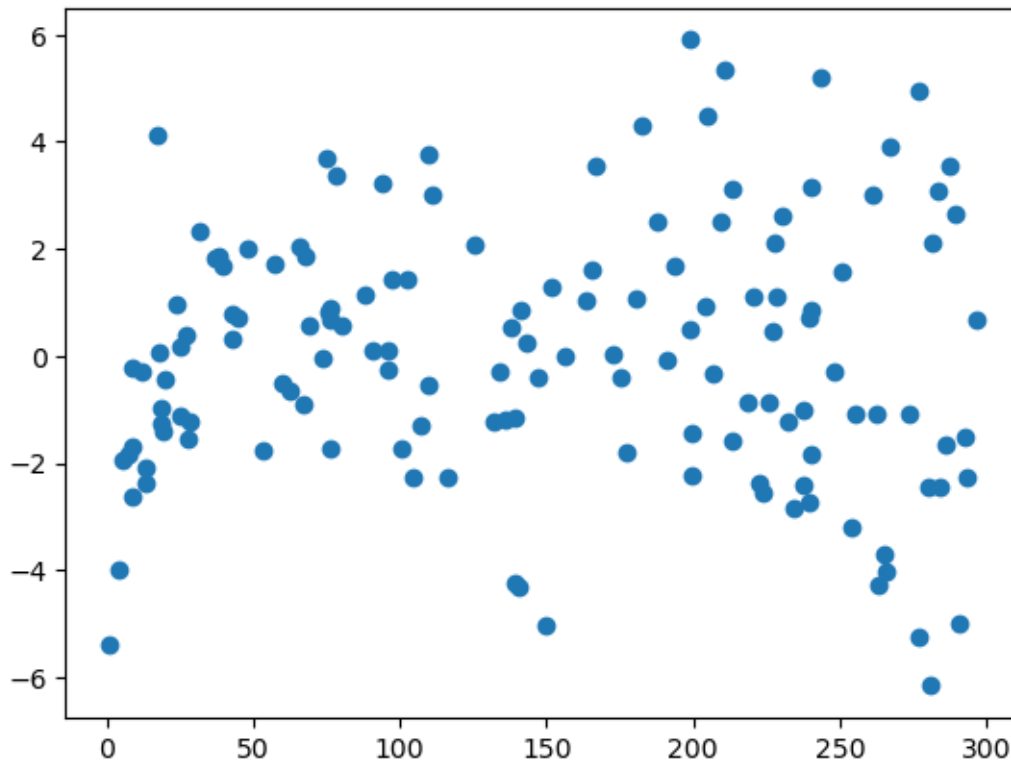
[38]: fig = plt.figure()
      sns.distplot(res, bins = 15)
      fig.suptitle('Error Terms', fontsize = 15)           # Plot heading
      plt.xlabel('y_train - y_train_pred', fontsize = 15)  # X-label
      plt.show()
```



The residuals are following the normally distributed with a mean 0. All good!

Looking for patterns in the residuals

```
[41]: plt.scatter(X_train, res)
plt.show()
```



We are confident that the model fit isn't by chance, and has decent predictive power. The normality of residual terms allows some inference on the coefficients.

Although, the variance of residuals increasing with X indicates that there is significant variation that this model is unable to explain.

As you can see, the regression line is a pretty good fit to the data

1.4 Step 5: Predictions on the Test Set

Now that you have fitted a regression line on your train dataset, it's time to make some predictions on the test data. For this, you first need to add a constant to the `X_test` data like you did for `X_train` and then you can simply go on and predict the y values corresponding to `X_test` using the `predict` attribute of the fitted regression line.

```
[45]: # Add a constant to X_test
X_test_sm = sm.add_constant(X_test)

# Predict the y values corresponding to X_test_sm
y_pred = lr.predict(X_test_sm)
```



```
[46]: y_pred.head()
```

```
[46]: 126      7.374140
      104     19.941482
      99     14.323269
      92     18.823294
      111    20.132392
      dtype: float64
```

```
[47]: from sklearn.metrics import mean_squared_error
      from sklearn.metrics import r2_score
```

Looking at the RMSE

```
[49]: #Returns the mean squared error; we'll take a square root
      np.sqrt(mean_squared_error(y_test, y_pred))
```

```
[49]: 2.019296008966232
```

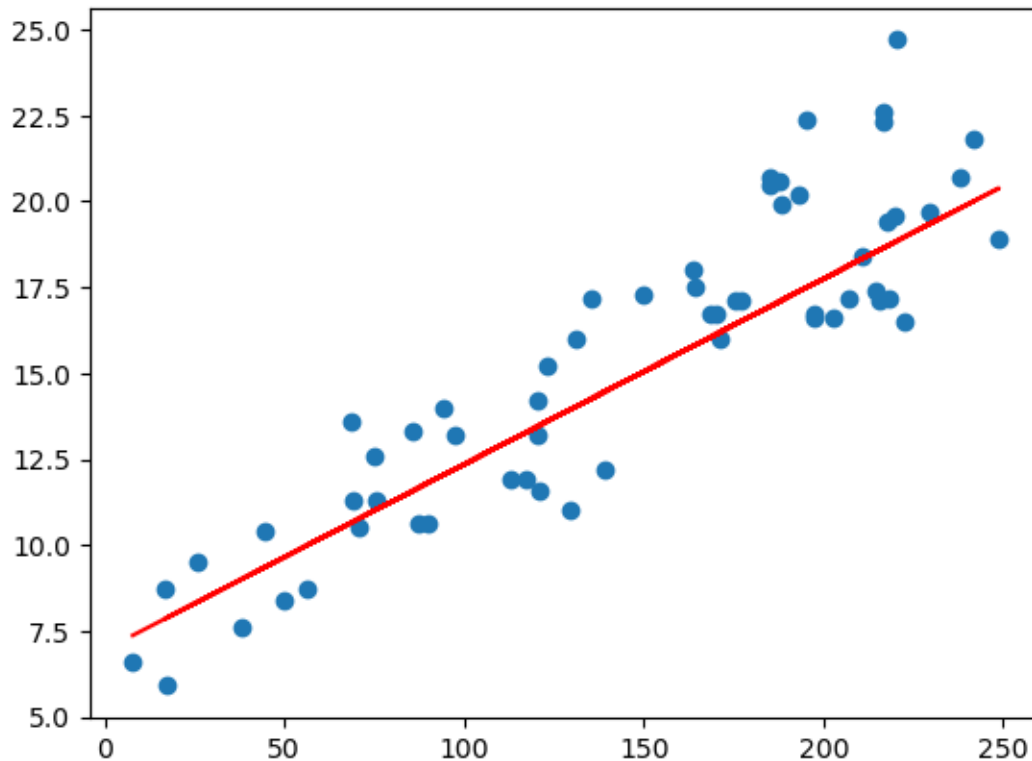
Checking the R-squared on the test set

```
[51]: r_squared = r2_score(y_test, y_pred)
      r_squared
```

```
[51]: 0.792103160124566
```

Visualizing the fit on the test set

```
[53]: plt.scatter(X_test, y_test)
      plt.plot(X_test, 6.948 + 0.054 * X_test, 'r')
      plt.show()
```



1.4.1 Linear Regression using `linear_model` in `sklearn`

Apart from `statsmodels`, there is another package namely `sklearn` that can be used to perform linear regression. We will use the `linear_model` library from `sklearn` to build the model. Since, we have already performed a train-test split, we don't need to do it again.

There's one small step that we need to add, though. When there's only a single feature, we need to add an additional column in order for the linear regression fit to be performed successfully.

```
[59]: from sklearn.model_selection import train_test_split
X_train_lm, X_test_lm, y_train_lm, y_test_lm = train_test_split(X, y,
    ↪ train_size = 0.7, test_size = 0.3, random_state = 100)
```

```
[60]: X_train_lm.shape
```

```
[60]: (140,)
```

```
[61]: X_train_lm = X_train_lm.values.reshape(-1,1)
X_test_lm = X_test_lm.values.reshape(-1,1)
```

```
[62]: print(X_train_lm.shape)
print(y_train_lm.shape)
print(X_test_lm.shape)
```

```
print(y_test_lm.shape)
```

```
(140, 1)
(140,)
(60, 1)
(60,)
```

```
[63]: from sklearn.linear_model import LinearRegression

# Representing LinearRegression as lr(Creating LinearRegression Object)
lm = LinearRegression()

# Fit the model using lr.fit()
lm.fit(X_train_lm, y_train_lm)
```

```
[63]: LinearRegression()
```

```
[64]: print(lm.intercept_)
print(lm.coef_)
```

```
6.948683200001357
[0.05454575]
```

The equation we get is the same as what we got before!

\$ Sales = 6.948 + 0.054* TV \$

Sklearn linear model is useful as it is compatible with a lot of sklearn utilities (cross validation, grid search etc.)

1.4.2 Q: Why is it called ‘R-squared’?

Based on what we learnt so far, do you see it? Can you answer this?

```
[74]: corrs = np.corrcoef(X_train, y_train)
print(corrs)
```

```
[[1.          0.90321277]
 [0.90321277  1.          ]]
```

```
[75]: corrs[0,1] ** 2
```

```
[75]: 0.8157933136480384
```

Correlation (Pearson) is also called “r” or “Pearson’s R”

1.4.3 Q: What is a good RMSE? Is there some RMSE that I should aim for?

You should be able to answer this by now!

Look at “Sharma ji ka beta”; he could answer this in a moment. How lucky is Sharma ji to have such a smart kid!

The RMSE: - depends on the units of the Y variables - is NOT a normalized measure

While it can't really tell you of the goodness of the particular model, it can help you compare models.

A better measure is R squared, which is normalized.

1.4.4 Q: Does scaling have an impact on the model? When should I scale?

While the true benefits of scaling will be apparent during future modules, at this juncture we can discuss if it has an impact on the model.

We'll rebuild the model after scaling the predictor and see what changes.

The most popular methods for scaling: 1. Min-Max Scaling 2. Standard Scaling

```
[88]: from sklearn.model_selection import train_test_split
      X_train, X_test, y_train, y_test = train_test_split(X, y, train_size = 0.7,
      ↪test_size = 0.3, random_state = 100)
```

SciKit Learn has these scaling utilities handy

```
[90]: from sklearn.preprocessing import StandardScaler, MinMaxScaler
```

```
[91]: # One aspect that you need to take care of is that the 'fit_transform' can be
      ↪performed on 2D arrays only. So you need to
      # reshape your 'X_train_scaled' and 'y_train_scaled' data in order to perform
      ↪the standardisation.
      X_train_scaled = X_train.values.reshape(-1,1)
      y_train_scaled = y_train.values.reshape(-1,1)
```

```
[92]: X_train_scaled.shape
```

```
[92]: (140, 1)
```

```
[93]: # Create a scaler object using StandardScaler()
      scaler = StandardScaler()
      # 'Fit' and transform the train set; and transform using the fit on the test set
      ↪later
      X_train_scaled = scaler.fit_transform(X_train_scaled)
      y_train_scaled = scaler.fit_transform(y_train_scaled)
```

```
[94]: print("mean and sd for X_train_scaled:", np.mean(X_train_scaled), np.
      ↪std(X_train_scaled))
      print("mean and sd for y_train_scaled:", np.mean(y_train_scaled), np.
      ↪std(y_train_scaled))
```

```
mean and sd for X_train_scaled: 2.5376526277146434e-17 0.9999999999999999
mean and sd for y_train_scaled: -2.5376526277146434e-16 1.0
```

```
[95]: # Let's fit the regression line following exactly the same steps as done before
X_train_scaled = sm.add_constant(X_train_scaled)

lr_scaled = sm.OLS(y_train_scaled, X_train_scaled).fit()
```

```
[96]: # Check the parameters
lr_scaled.params
```

```
[96]: array([-2.44596010e-16,  9.03212773e-01])
```

As you might notice, the value of the parameters have changed since we have changed the scale. Let's look at the statistics of the model, to see if any other aspect of the model has changed.

```
[99]: print(lr_scaled.summary())
```

```

                                OLS Regression Results
=====
Dep. Variable:                  y      R-squared:                  0.816
Model:                        OLS      Adj. R-squared:             0.814
Method:                    Least Squares  F-statistic:                 611.2
Date:                Wed, 02 Oct 2024  Prob (F-statistic):       1.52e-52
Time:                  04:01:22      Log-Likelihood:            -80.233
No. Observations:                140      AIC:                   164.5
Df Residuals:                    138      BIC:                   170.3
Df Model:                        1
Covariance Type:                nonrobust
=====

```

	coef	std err	t	P> t	[0.025	0.975]
const	-2.446e-16	0.037	-6.69e-15	1.000	-0.072	0.072
x1	0.9032	0.037	24.722	0.000	0.831	0.975

```

=====
Omnibus:                        0.027      Durbin-Watson:             2.196
Prob(Omnibus):                  0.987      Jarque-Bera (JB):           0.150
Skew:                          -0.006      Prob(JB):                   0.928
Kurtosis:                      2.840      Cond. No.                   1.00
=====

```

Notes:

[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.

Model statistics and goodness of fit remain unchanged.

So why scale at all?

- Helps with interpretation (we'll be able to appreciate this better in later modules)
- Faster convergence of gradient descent