Linear Regression with Gradient Descent

Article · June 2020

CITATIONS
5

READS
4,431

1 author:

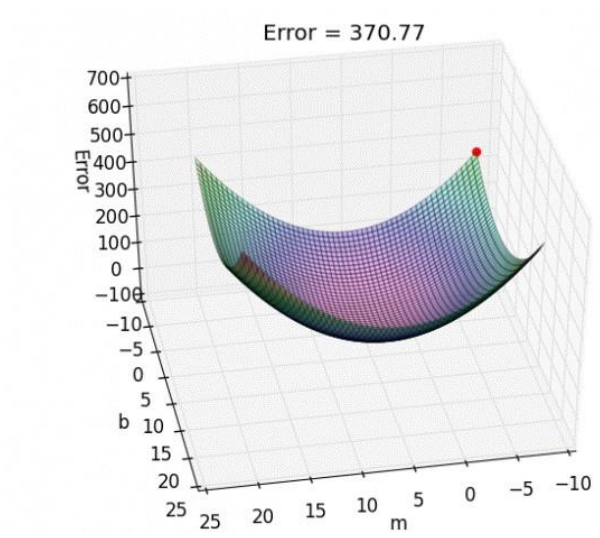
Rukshan Manorathna
University of Colombo
16 PUBLICATIONS **11** CITATIONS

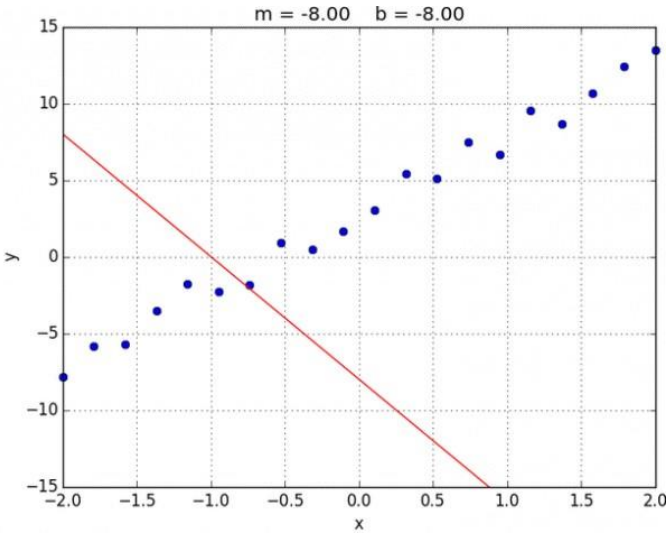
SEE PROFILE

Linear Regression with Gradient Descent

Understand how linear regression really works behind the scenes

Rukshan Pramoditha
Jun 14 2020





You are ***ALL*** welcome to another exciting tutorial at **Data Science 365**! So far I've discussed the fundamentals of **Data Science**, **Machine Learning** and various Python libraries (modules/packages) such as **numpy**, **pandas**, **matplotlib**, **seaborn** which can be used for your data analysis task.

It's time to practically apply all these things that I've discussed so far at Data Science 365. I highly recommend you to read my previous articles published at https://medium.com/data-science-365 before reading this one. Today, in this tutorial, I will discuss the most fundamental Machine Learning algorithm called **Linear Regression** by following the steps of the Predictive Analytics process. By the end of this tutorial, you can master the complete linear regression process from problem definition to model implementation.

A short introduction to predictive analytics

Predictive Analytics is an applied field that uses a variety of quantitative methods that make use of data in order to make predictions. It often involves solving problems. The steps of the predictive analytics process are:

1. **Problem understanding and definition**
2. **Data collection and preparation**
3. **Dataset understanding using Exploratory Data Analysis (EDA)**
4. **Model building**
5. **Model evaluation**
6. **Model implementation**

The predictive analytics process is **not a one-time process**. It is an **iterative process** in which you will always find yourself going back and forth between the above steps to build a better model which captures a great amount of variability in your data.

Now, let me start building the linear regression model by following the steps of the predictive analytics process.

Problem understanding and definition

This is the first step in the process. *The goal is to understand the problem and how the potential solution would look from the business perspective*.

The problem is: *Build a Simple Linear Regression Model to predict sales based on the money spent on TV advertising*.

When we consider the model in the business perspective, it should add some value to the business in terms of money. By using the various output values of the model, the company should be able to decide whether TV advertising is effective or not and if it is effective, how much money the company should spend on TV advertising to get a particular increase of sales.

The model that I build to solve the above problem is called a **Regression Model** because it predicts a continuous-valued output (**Classification Model**, by contrast, predicts a discrete-valued output — 0 or 1).

This regression model involves two variables.

1. The variable that we are interested in predicting is called the target variable (=response variable/dependent variable/outcome variable). The variable **sales** is the target variable in our model. The outputs of the model, called **predictions**, are the values of this variable for given inputs.

2. The variable that we use to predict the values of the target variable is called the explanatory variable (=feature/attribute/independent variable/predictor/regressor). The variable **TV ad spending** is the explanatory variable in our model. The inputs of the model are the values of this variable.

If there is a **linear relationship** between the feature and the response variable, the regression model is called the **linear regression model**. We can confirm this by visualizing the data in a scatterplot. This will be done in the EDA section of this tutorial.

Since this linear regression model has only one feature (explanatory variable), it is called a **Simple Linear Regression Model** (By contrast, a **Multiple Linear Regression Model** uses multiple features to make a prediction). This linear regression model is also a **Univariate Linear Regression Model** since we are only trying to predict only one target variable (By contrast, a **Multivariate Linear Regression Model** predicts multiple correlated target variables).

In this tutorial, I will take a machine learning approach to linear regression. In machine learning, the linear regression is a supervised learning algorithm in which you use both input and output variables to learn the mapping function from the input to the output, y=f(x). **The goal is to determine the mapping function so well that when you have new input data (x), you can predict the output for that data accurately**. I will discuss three different approaches to find the optimized values for the model parameters.

1. **Linear Regression with Gradient Descent**
2. **Linear Regression with the Scikit-learn LinearRegression estimator**
3. **Linear Regression with the Normal Equation**

In the first approach, I'll use Python and its numpy module along with the gradient descent algorithm to find the optimized values for the model parameters. In the second approach, I'll use the popular Scikit-learn Machine Learning library. So, we can train the model very easily. In the third approach, I'll use the normal equation along with the linear algebra and numpy ndarray objects to find the optimized values for the parameters analytically.

There is almost no difference after training. All these algorithms end up with very similar models and make predictions in exactly the same way.

I will use the **Root Mean Squared Error (RMSE)** and the **R-squared (R^2coefficient of determination)** as the key metrics of model performance to evaluate the regression model.

I will also use various graphical techniques such as **prediction error plot**, **residual plot** and **distribution of residuals** to evaluate the model and verify its assumptions.

Data collection and preparation

The goal is to get a dataset that is ready for analysis. Luckily, the dataset is readily available. It is in the **.csv** format. So, we can use the pandas **read_csv()** function passing the name **'Advertising.csv'** to load the dataset. Then, we use the DataFrame **head()** method to see the first 5 rows of the dataset.

```
import pandas as pd

df = pd.read_csv('Advertising.csv')
df.head()
```

	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	9.3
3	151.5	41.3	58.5	18.5
4	180.8	10.8	58.4	12.9

The first 5 rows of the Advertising dataset

The Advertising dataset captures **sales** revenue generated with respect to advertisement spends across multiple channels like **radio**, **TV** and **newspaper**.

As you can see, there are four columns in the dataset. Since our problem definition involves only **sales** and **TV** columns in the dataset, we do not need **radio** and **newspaper** columns (In a separate article, I will discuss how to build a multiple linear regression model using all these variables. In this tutorial we just consider only **sales** and **TV** variables according to our problem definition). So, I remove the unnecessary variables in place so that the original DataFrame is modified.

```
df.drop(columns=['radio', 'newspaper'], inplace=True)
```

If you call the **head()** method again, you can see that the DataFrame (df) now consists of only **TV** and **sales** columns.

```
df.head()
```

	TV	sales
0	230.1	22.1
1	44.5	10.4
2	17.2	9.3
3	151.5	18.5
4	180.8	12.9

Advertising dataset with two columns that we're interested in

The shape of the modified advertising dataset is:

```
df.shape
```

```
(200, 2)
```

The dataset now has 2 columns and 200 rows (observations). The two columns that we're interested in are:

- **TV:** the amount of money spent on TV advertising — in US dollars
- **sales:** the sales revenue generated with respect to advertisement spends — in 1000s of US dollars

The next step is to check the data type of each variable. For this, we can use the DataFrame **info()** method.

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 200 entries, 0 to 199
Data columns (total 2 columns):
 #   Column  Non-Null Count  Dtype  
---  ------  --------------  -----  
 0   TV      200 non-null    float64
 1   sales   200 non-null    float64
dtypes: float64(2)
memory usage: 3.2 KB
```

Scikit-learn machine learning library only works with variables that are numeric. If there are variables with string values, you need to encode them into numeric values. Fortunately, the dataset contains all numerical values so that no encoding is necessary.

Next, we need to check to see if there are any missing values. To do so, use DataFrame **isnull()** method combined with **sum()**.

```
df.isnull().sum()
```

```
TV       0
sales    0
dtype: int64
```

Again, the dataset is good, as it does not have any missing values.

That's it. Now, the dataset is ready for the analysis. For future reference, I save it as an Excel file. I use **to_excel()** method of pandas DataFrame. By setting **index=False**, the row index labels are not saved in the Excel file.

```
df.to_excel('Advertising.xlsx', index=False)
```

After running this line of code, the Excel file will be saved in your current working directory.

Dataset understanding using Exploratory Data Analysis (EDA)

The goal is to understand our dataset. The dataset is ready. It is time to start understanding it using Exploratory Data Analysis (EDA).

There are two types of EDA techniques:

- **Numerical calculations**
- **Visualizations**

It is a good practice to use these two types of techniques **simultaneously** to get a better understanding of different characteristics of our dataset, its variables and the potential relationship between them.

Analyse the relationship between the predictor and the target variable

For this, we have two fundamental standard tools:

- **The scatterplot for visualization**
- **Pearson correlation coefficient as a numerical calculation**

The scatterplot is super useful to visualize the relationship between the predictor and the target variable. It is simply obtained by plotting pairs of points, each having the value of the predictor determining the position on the x-axis and the value of the target variable determining the position on the y-axis. If there is a relationship between the variables, we can look for the following characteristics:

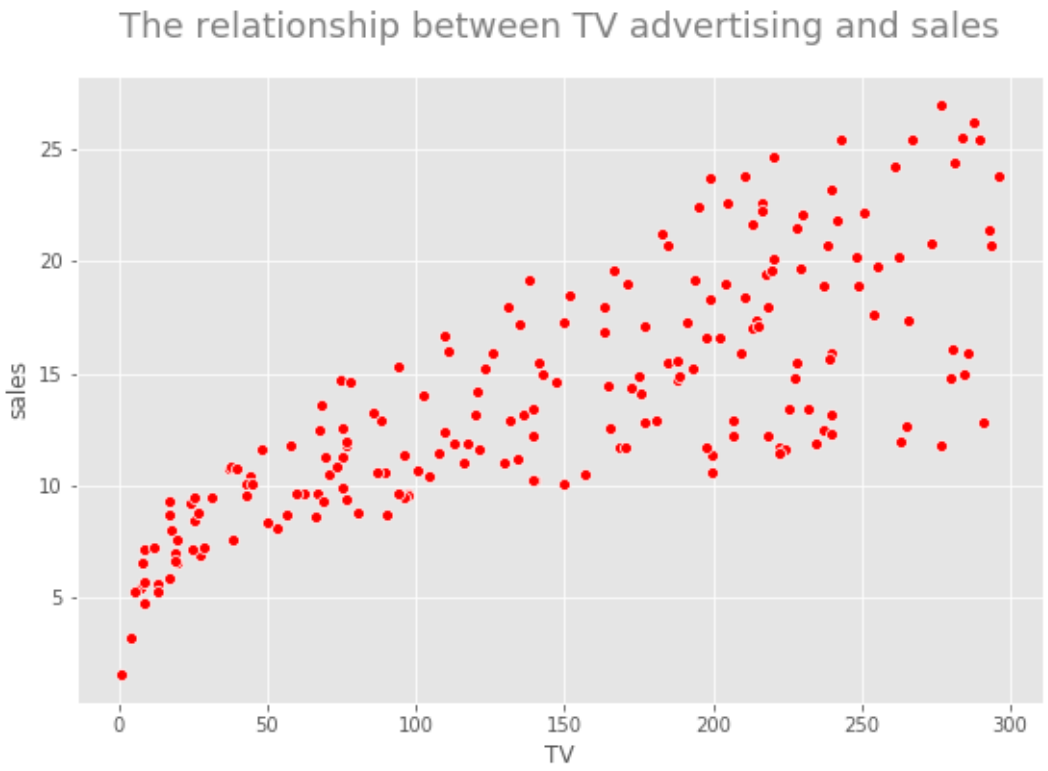
Overall pattern: This can be a linear pattern, a curvy pattern, an exponential pattern or a more complicated one.

Strength: This refers to how closely the points follow the pattern.

Direction: This can be either positive or negative. **Positive** means that when one variable increases, the other also tends to increase and vice versa. In this case, the pattern goes upward. **Negative** means that when one variable increases, the other tends to decrease and vice versa. In this case, the overall pattern goes downward.

The Pearson correlation coefficient is a numerical indication of the strength of the **linear association** between two numerical variables. If the relationship between the variables is nonlinear, then this correlation coefficient could be misleading. Be careful when using this! It is always a good idea to take a look at both the scatterplot and the correlation coefficient simultaneously.

```
%matplotlib inline

import seaborn as sns
import matplotlib.pyplot as plt

plt.style.use('ggplot')

fig, ax = plt.subplots(figsize=(9, 6))

sns.scatterplot(x='TV', y='sales', data=df, ax=ax, color='red')

ax.set_title('The relationship between TV advertising and sales', pad=20, size=18, color='gray')
plt.savefig('scatterplot.png')
```



There seems to be a **positive linear relationship** between TV advertising and sales. Now, I calculate the Pearson correlation coefficient between the two variables to get an idea about the strength of the linear relationship.

```
df.corr(method='pearson')
```

	TV	sales
TV	1.000000	0.782224
sales	0.782224	1.000000

The value is approximately **0.78** which indicates that **TV Ad Spending is highly correlated with sales**.

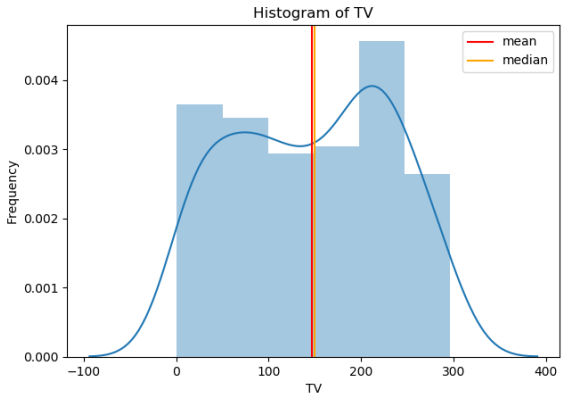
Since there seems to be a **linear relationship** between TV advertising and sales, we can build a linear regression model as our problem statement defines. What do you do if you see a non-linear relationship between the two variables? You cannot build a linear regression model. So, you may go back to step 1 and reframe the problem definition or you may stop the process by giving the reasons. But do not build a linear regression model for variables which have a non-linear relationship!

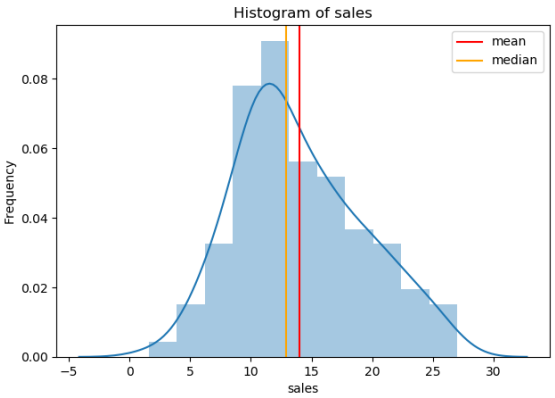
Analyse the distribution of variables

The default plot used to get a graphical representation of a continuous variable is the histogram. This plot that shows the distribution of the values of a numerical variable by plotting a series of non-overlapping, continuous bars: the values of the variable on the x-axis and the frequency of observations on the y-axis. Thus, this graph tells us which ranges of values are more and less frequent.

Now, I create two histograms for **TV** and **sales** variables in the same figure.

```
%matplotlib inline

import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt

fig, ax = plt.subplots(nrows=1, ncols=2, figsize=(14, 5.4))
plt.tight_layout(pad=5)

sns.distplot(df['TV'], ax=ax[0])
ax[0].axvline(x=np.mean(df['TV']), color='red', label='mean')
ax[0].axvline(x=np.median(df['TV']), color='orange', label='median')
ax[0].set_ylabel('Frequency')
ax[0].set_title('Histogram of TV')
ax[0].legend(loc='upper right')

sns.distplot(df['sales'], ax=ax[1])
ax[1].axvline(x=np.mean(df['sales']), color='red', label='mean')
ax[1].axvline(x=np.median(df['sales']), color='orange', label='median')
ax[1].set_ylabel('Frequency')
ax[1].set_title('Histogram of sales')
ax[1].legend(loc='upper right')

plt.savefig('histograms.png', dpi=100)
```





I also calculate the summary statistics for each variable with just one line of code!

```
df.describe().T
```

	count	mean	std	min	25%	50%	75%	max
TV	200.0	147.0425	85.854236	0.7	74.375	149.75	218.825	296.4
sales	200.0	14.0225	5.217457	1.6	10.375	12.90	17.400	27.0

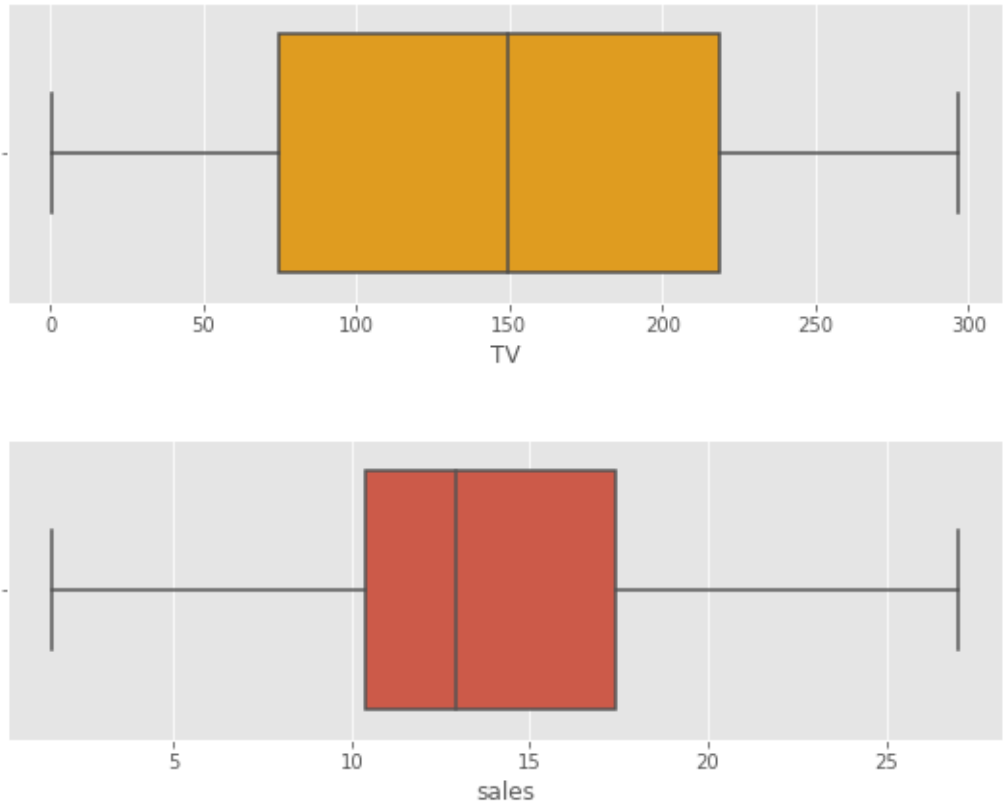
Now, I interpret the above two histograms:

- **Histogram of TV Ad spending:** The distribution of TV Ad spending is symmetric. The mean and median are very close in value. TV Ad spending follows a uniform distribution with a mean of 147. The center is close to 150. The typical deviation in TV Ad spending from the mean is about 85.9 units. There is a large variability in TV Ad spending. We cannot see observations outside the overall pattern.

- **Histogram of sales:** The distribution of sales is single-peaked (unimodal) and symmetric. The mean and median are close in value. Sales approximately follow a normal distribution with a mean of 14.02. The center is close to 13. The typical deviation in sales from the mean is about 5.2 units. Variability in sales is not very large. There are no observations outside the overall pattern.

Another type of graph that we create is boxplot. A boxplot provides a graphical representation of 5-Number Summary — MINIMUM, Q1, Median, Q3 and MAXIMUM.

```
%matplotlib inline

import seaborn as sns
import matplotlib.pyplot as plt

plt.style.use('ggplot')

fig, ax = plt.subplots(nrows=2, ncols=1, figsize=(9, 7))
plt.tight_layout(pad=5)

sns.boxplot(x=df['TV'], ax=ax[0], color='orange')
sns.boxplot(x=df['sales'], ax=ax[1])

plt.savefig('boxplots.png')
```



Boxplots are very useful in detecting outliers. As you can see in the above boxplots, there are no outliers present in our data. This is because there are no data points beyond the range of 1.5*IQR from Q1 and Q3.

Model building and model evaluation

The goals are to build the simple linear regression model that solves the problem, determine how good the model is in providing the solution and verify the assumptions of the model.

The general form for a simple linear regression model

$$y = h_\theta(x) = \theta_0 + \theta_1 x_1 + \epsilon$$

- **θ0**, **θ1:** Model parameters
- **y:** Response variable or **$h\theta(x)$:** Hypothesis function
- **x1:** Predictor
- **ϵ:** Error term (Residual)

Linear regression assumptions

- The predictors and response are **linearly** related — We've checked this assumption in the EDA section. Now, it is OK!
- The residuals (observed values-predicted values) are approximately normally distributed with the mean 0 and a fixed standard deviation — We'll verify this assumption by drawing a histogram of residuals.
- The residuals are uncorrelated or independent — We'll verify this assumption by drawing a residual plot.

The cost function (Mean squared error function)

$$J(\theta_0, \theta_1) = \frac{1}{2m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)} \right)^2$$

- **m:** Number of training examples
- **x(i):** x-value of ith training example
- **y(i):** y-value of ith training example
- **J(θ0 , θ1):** Cost function

To break it apart, you can think this equation as 1/2 * (x-bar) where x-bar is mean of the squares of the difference between the predicted values and the actual values. The 1/2 term is for convenience for the computation of the gradient descent, as the derivative term of the square function will cancel out the 1/2 term.

The goal is to minimize the above cost function so that we can find the optimized values for the model parameters. By using those values, we can fit the best possible straight line to the data.

In this tutorial, I will discuss three different approaches to find the optimized values for the model parameters.

Approach 1: Linear Regression with Gradient Descent

Here I use Python and its numpy module along with the gradient descent algorithm to find the optimized values for the model parameters. Here we do not use the popular Scikit-learn Machine Learning library. So, this is a great way to have a deeper understanding of behind the scene process or the inner machinery of Linear Regression.

The gradient descent algorithm

The gradient descent algorithm is an optimization algorithm that can be used to minimize the above cost function and find the optimized values for the linear regression model parameters. This can also be applied for more general problems like multiple linear regression problems with 100 or 1000 predictors, although we use it here with just one predictor. For just two parameters (as in simple linear regression), the gradient descent algorithm is:

$$\text{repeat until convergence } \{$$
$$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta_0, \theta_1)$$
$$(\text{for } j = 1 \text{ and } j = 0)$$
$$\}$$

:= notation: Assignment operator. In programming, it is just = notation.

***α*:** Learning rate. It is a fixed value.

Derivative term: The partial derivative of the cost function J(θ0 , θ1) for j=0 and 1.

After we partially differentiate the cost function J(θ0 , θ1) for j=0 and 1, the algorithm is:

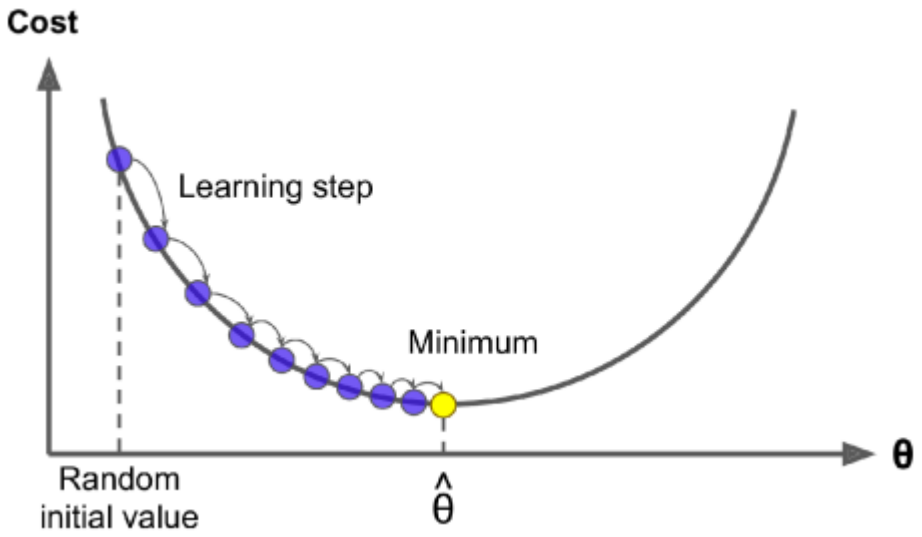
$$\text{repeat until convergence } \{$$
$$\theta_0 := \theta_0 - \alpha \frac{1}{m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)} \right)$$
$$\theta_1 := \theta_1 - \alpha \frac{1}{m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)} \right) \cdot x^{(i)}$$
$$\}$$

Now we follow the below steps to implement the gradient descent to our simple linear regression model.

Step 1: Start with some initial guesses for $\theta 0$, $\theta 1$ (usually $\theta 0 = 0$, $\theta 1 = 0$).

Step 2: Choose a good value for $\boldsymbol{\alpha}$.

Step 3: Simultaneously update the values of $\theta 0$, $\theta 1$ to reduce the cost function J($\theta 0$, $\theta 1$) until we hopefully end up at the minimum.



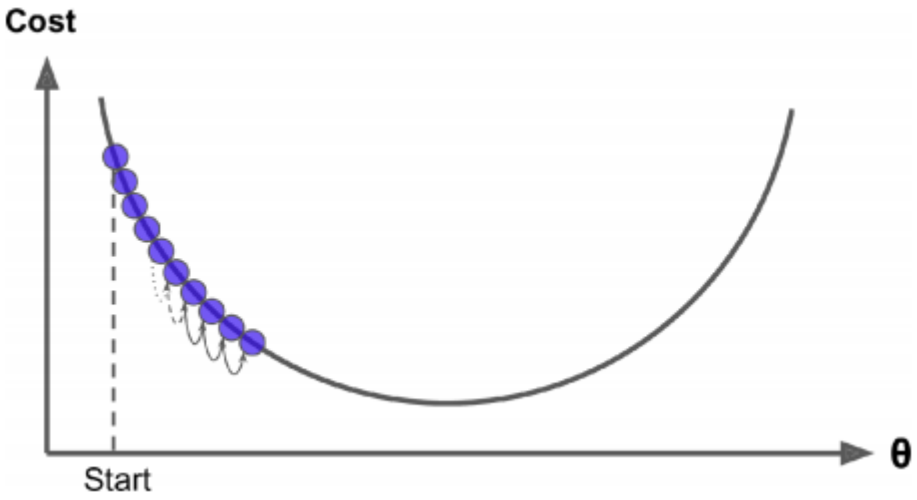
Now you may have questions like these:

- How do I choose a good value for $\boldsymbol{\alpha}$?
- How do I know whether the algorithm ends up at the minimum?

Choosing α (Learning rate)

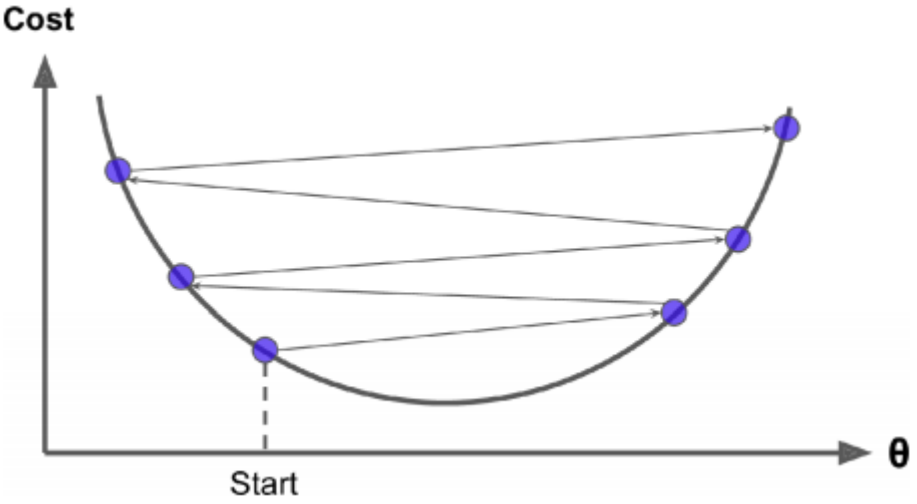
we should adjust the value of $\boldsymbol{\alpha}$ to ensure that the gradient descent algorithm converges in a reasonable time. Failure to converge or too much time to obtain the minimum value implies that our $\boldsymbol{\alpha}$ is wrong.

If $\boldsymbol{\alpha}$ is too small, the gradient descent can be slow. It will require more iterations (hence time) to reach the minimum.



The learning rate is too small

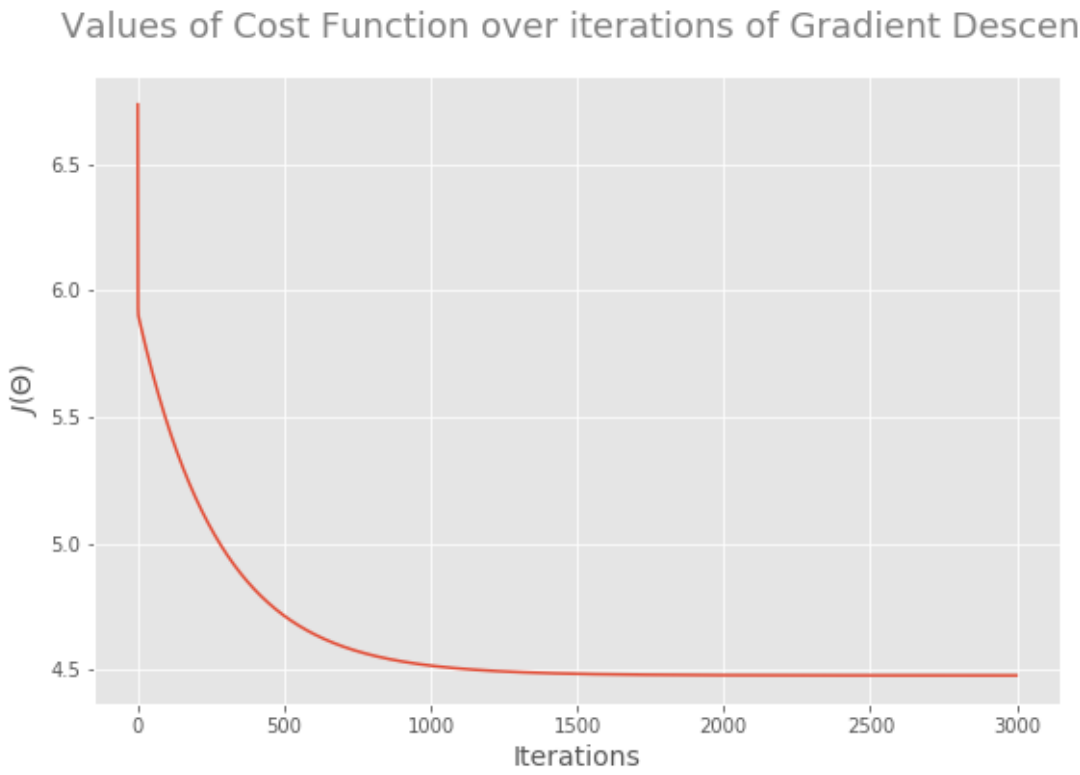
If $\boldsymbol{\alpha}$ is too large, the gradient descent can overshoot the minimum. In that case, it may fail to converge or even diverge. If this happens when you run your Python code, NaN values are returned for $\theta 0$, $\theta 1$. In that case, you should decrease the value of $\boldsymbol{\alpha}$ and run the algorithm again.



The learning rate is too large

Confirming that the algorithm ends up at the minimum

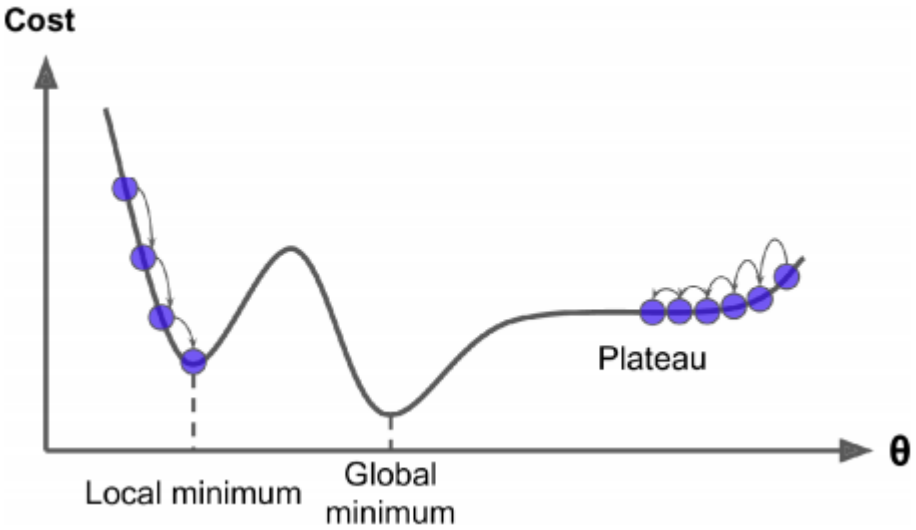
If you choose a good value for $\boldsymbol{\alpha}$ and run the algorithm enough iterations, the algorithm will end up at the minimum. The best way to confirm this is to plot the values of cost function over iterations of gradient descent. You should see a **monotonically decreasing function** (a function that is always decreasing or remaining constant and never increasing) like this:



At the minimum, the derivative will always be 0 and thus the value of the cost function will not be changed. The cost at the last iteration is the minimum cost and the parameter values associated with that cost are the optimized $\boldsymbol{\theta}$ values.

Global minimum and local minimum

Many cost functions have a local minimum as well as a global minimum as shown in the following image:



Fortunately, the MSE cost function for a Linear Regression model happens to be a convex function which has just one global minimum, but no local minima. So, Gradient Descent is guaranteed to approach the global minimum (if you wait long enough and if the learning rate is not too high).

Feature scaling

If the predictors have very different scales, you must do feature scaling before running the gradient descent algorithm. If you run the algorithm without feature scaling, it will take a very long time to reach the global minimum.

1. No need to do feature scaling if there is only one predictor variable.

2. Do feature scaling if predictors have very different scales such as 1:100, 1:1000.

3. Do feature scaling only for predictors. No need to apply feature scaling for the response variable even if it has a very different scale.

4. A common technique for feature scaling is **mean normalization** which involves subtracting the average value for an input variable and divide that result by the standard deviation of that variable.

$$x_i := \frac{x_i - \mu_i}{s_i}$$

That's it. You have just learned two algorithms in Machine Learning: **Simple Linear Regression** and **Gradient Descent**. Now, It is time to implement those algorithms to our problem by writing Python codes. Good knowledge of Python and numpy library is highly recommended to understand the process.

Step 1: Define X, y, theta and m

```
# Number of training examples
m = df['sales'].values.size

# X is the feature matrix.
# We add another column to accomodate the intercept term and
# set its values to ones
# X.shape = (200, 2), 2-d array
X = np.append(np.ones((m, 1)), df['TV'].values.reshape(m, 1), axis=1)

# y is the target vector
# y.shape = (200, 1), 2-d array
y = df['sales'].values.reshape(m, 1)

# theta is the parameter matrix
# theta.shape = (2, 1), 2-d array
theta = np.zeros((2,1))
```

Step 2: Compute the Cost $J(\theta)$

Now, I define a function called **cost_function** which takes the inputs **X**, **y** and **theta** and returns the cost.

```
def cost_function(X, y, theta):
    # Prediction = Feature Matrix x Parameter Matrix
    # Matrix Multiplication
    y_pred = np.dot(X, theta)
    sqrd_error = (y_pred - y) ** 2
    cost = 1 / (2 * m) * np.sum(sqrd_error)
    return cost
```

Now I calculate cost when all the parameters are zero (when parameter matrix **theta** contains the elements all zero).

```
cost_function(X, y, theta)
```

```
111.858125
```

The cost is approximately 111.86. Mathematically, this is the cost associated with the line which coincides with the x-axis. This line is not the fitted regression line. From the scatterplot that I created before, the fitted regression line should have a positive slope ($\theta 1 > 0$). So, we can further minimize the cost by adjusting the values of model parameters.

Step 3: Run the Gradient Descent

First, I define a function called **gradient_descent()** which takes the inputs **X**, **y**, **theta**, **alpha**, **iterations** and returns the **theta** and **costs** array.

```
def gradient_descent(X, y, theta, alpha, iterations):
    costs = []
    for i in range(iterations):
        y_pred = np.dot(X, theta)
        der = np.dot(X.transpose(), (y_pred - y)) / m
        theta -= alpha * der
        costs.append(cost_function(X, y, theta))

    return theta, costs
```

Now, I run the gradient descent.

```
theta, costs = gradient_descent(X, y, theta, alpha=0.000068, iterations=400000)
```

```
theta
```

```
array([[7.02543044],
       [0.04757302]])
```

The cost associated with the above theta values is:

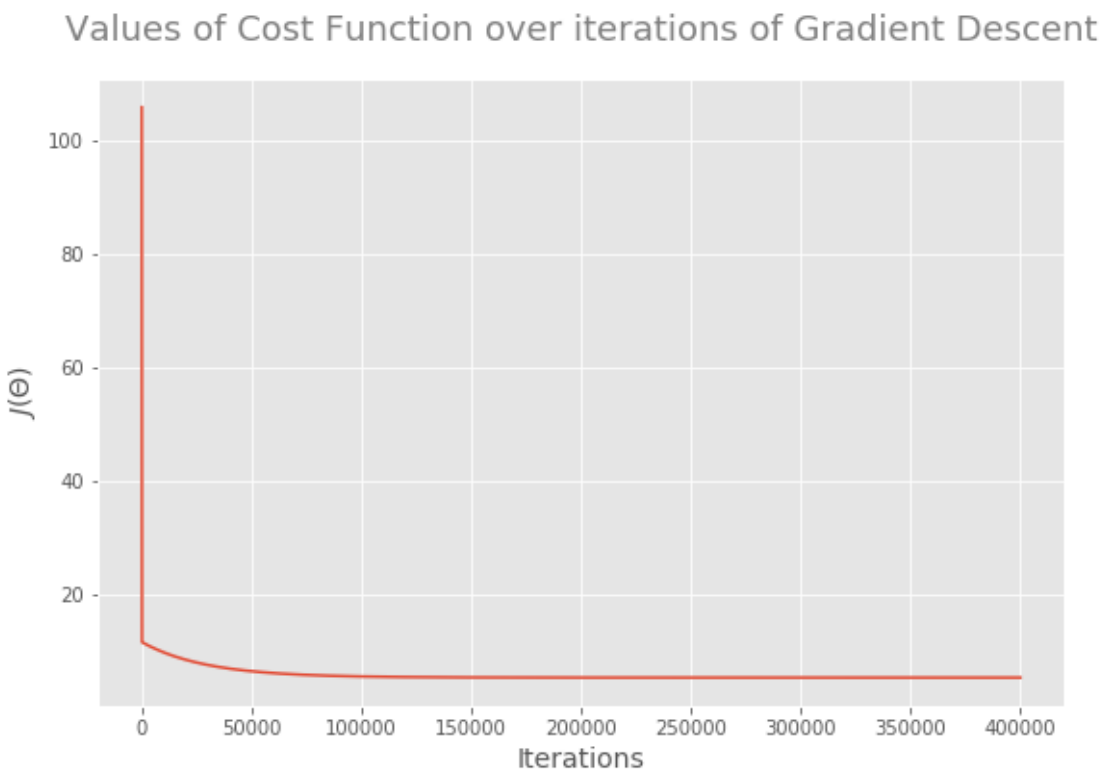
```
# See the last element of the cost array
# This is the minimum cost

costs[-1]
```

```
5.256332955936214
```

If the value of the alpha is not too high and if we run the algorithm enough iterations, the returned theta values are the optimized values for the model parameters and the last element of the returned costs array is the minimum cost. We can confirm this from a graphical visualization.

Step 4: Plot the Convergence

```
fig, ax = plt.subplots(figsize = (9, 6))

ax.plot(costs)

ax.set_title("Values of Cost Function over iterations of Gradient Descent",
             pad=20, size=18, color='gray')
ax.set_xlabel("Iterations", size=14)
ax.set_ylabel("$J(\Theta)$", size=14)
plt.savefig('cost vs iterations.png')
```



So, Gradient Descent has reached the global minimum. The returned theta values are the optimized values for the model parameters and the minimum cost is about 5.26.

Step 5: Train the model for making predictions

Now, we can use the optimized values of theta to train the model for making predictions. First, I round the parameter values to the 3rd decimal place.

```
theta
```

```
array([[7.02543044],
       [0.04757302]])
```

```
# Round theta values to the 3rd decimal place
np.round(theta, 3)
```

```
array([[7.025],
       [0.048]])
```

$\theta_0 = 7.025$
$\theta_1 = 0.048$

$$sales = 7.025 + 0.048 \times TV$$

The interpretation for $\boldsymbol{\theta 1}$ is: For a $1 increase in TV advertising spend, the sales will increase by $48 (this is because sales are in 1000s of US dollars).

The interpretation for $\boldsymbol{\theta 0}$ is: Without any TV advertising spend, a particular product will have a sale of $7025 (this is because sales are in 1000s of US dollars).

```
y_pred = np.dot(X, np.round(theta, 3))
dic = {'sales (Actual)':y.flatten(),
       'sales (Predicted)':np.round(y_pred, 1).flatten()}
df1 = pd.DataFrame(dic)
```

The first 5 observations are:

```
df1.head()
```

	sales (Actual)	sales (Predicted)
0	22.1	18.1
1	10.4	9.2
2	9.3	7.9
3	18.5	14.3
4	12.9	15.7

The last 5 observations are:

```
df1.tail()
```

	sales (Actual)	sales (Predicted)
195	7.6	8.9
196	9.7	11.5
197	12.8	15.5
198	25.5	20.6
199	13.4	18.2

It works, although the predictions are not exactly accurate.

To make a prediction for a value that is not in our dataset, we can use the **predict()** function which is a user-defined function. This function is user friendly because you need to just pass the x-value to it and the function will do the rest for you.

```
theta
```

```
array([[7.02543044],
       [0.04757302]])
```

$$y = x.\theta$$

```
def predict(a):
    x = np.array([1, a]).reshape(1, 2)
    y_pred = np.dot(x, theta) # 2d-array
    return y_pred[0,0]
```

```
# Pass a float to predict()
predict(30)
```

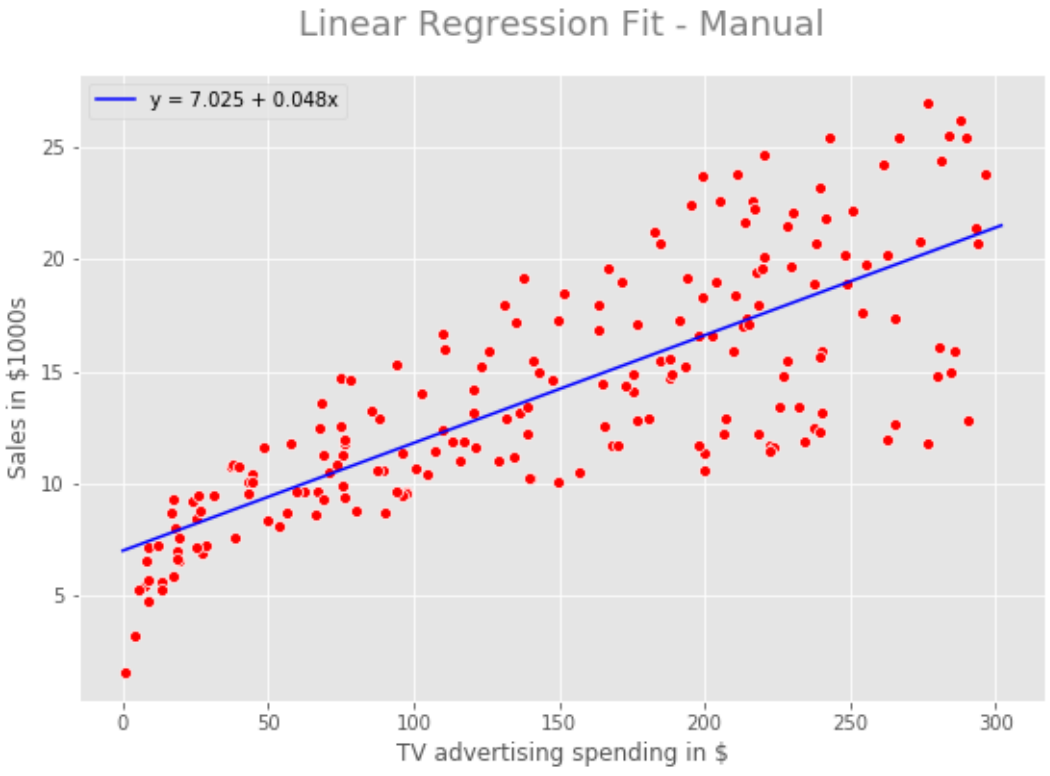
```
8.452620937358214
```

The predicted sales for the TV advertising spending of $30 is approximately 8.45 y units.

Step 6: Fit the regression line

Now, I fit the regression line manually.

```
%matplotlib inline

import seaborn as sns
import matplotlib.pyplot as plt

plt.style.use('ggplot')

fig, ax = plt.subplots(figsize=(9, 6))
sns.scatterplot(x='TV', y='sales', data=df, ax=ax, color='red')

a = np.round(theta, 3)
x_value = np.array(range(0, 303))
y_value = a[0, 0] + (a[1, 0] * x_value)

sns.lineplot(x_value,y_value, ax=ax, label='y = 7.025 + 0.048x', color='blue')

ax.set_title('Linear Regression Fit - Manual', pad=20, size=18, color='gray')
ax.set_xlabel('TV advertising spending in $')
ax.set_ylabel('Sales in $1000s')
ax.legend(loc='upper left')
plt.savefig('linear regerssion fit.png')
```



Step 7: Evaluate the model performance

Now, I use the **Root Mean Squared Error (RMSE)** and the **R-squared (R^2coefficient of determination)** as the key metrics of model performance to evaluate the regression model.

With only one predictor, $\mathbf{R^2 = r^2}$, where r is the Pearson correlation coefficient between **sales** and **TV advertising**.

The form for the **Root Mean Squared Error (RMSE)** is:

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

RMSE is the square root of the **Mean Squared Error (MSE)**. The **RMSE** can be interpretable in y units. Smaller values for **RMSE** are better and the best value we can get (the perfect model) is 0.

```
R_squared = 0.78 ** 2
MSE = ((y - y_pred) ** 2).sum() / m
RMSE = np.sqrt(MSE)
print('R^2: ', np.round(R_squared, 2))
print('RMSE: ', np.round(RMSE, 2))
```

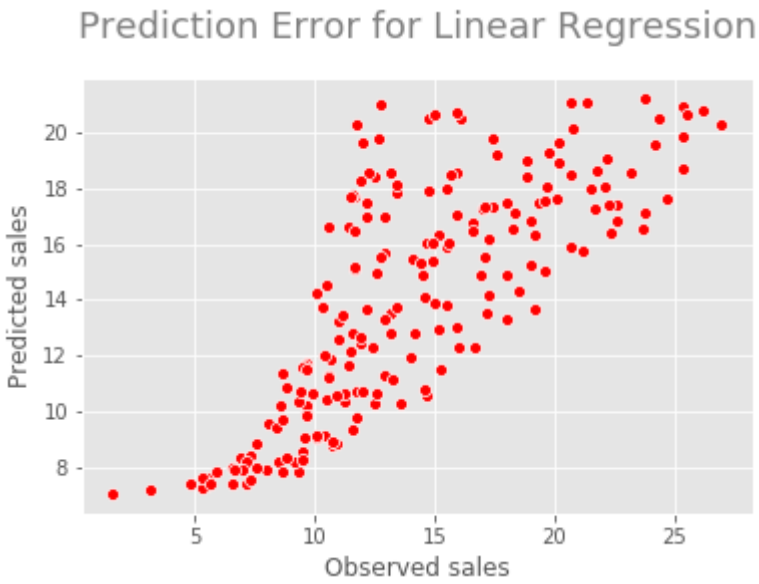
```
R^2:  0.61
RMSE:  3.24
```

R^2 value is 0.61. It means that 61% of the variability observed in the sales is explained or captured by our model and the other 39% is due to some other factors and, of course, randomness! With R^2=0.61, the model is not actually very good. This means we have room for improvement of the model by including multiple predictors.

RMSE value is 3.24. It means that on average, the predictions of the model are 3.24 units away from the actual values. 3.24 is a better value for our model.

As a graphical method, we can use a prediction error plot to evaluate the regression model.

```
fig, ax = plt.subplots()

sns.scatterplot(x=y.flatten(), y=y_pred.flatten(), ax=ax, color='red')

ax.set_title('Prediction Error for Linear Regression', pad=20, size=18, color='gray')
ax.set_xlabel('Observed sales')
ax.set_ylabel('Predicted sales')
plt.savefig('prediction error.png')
```



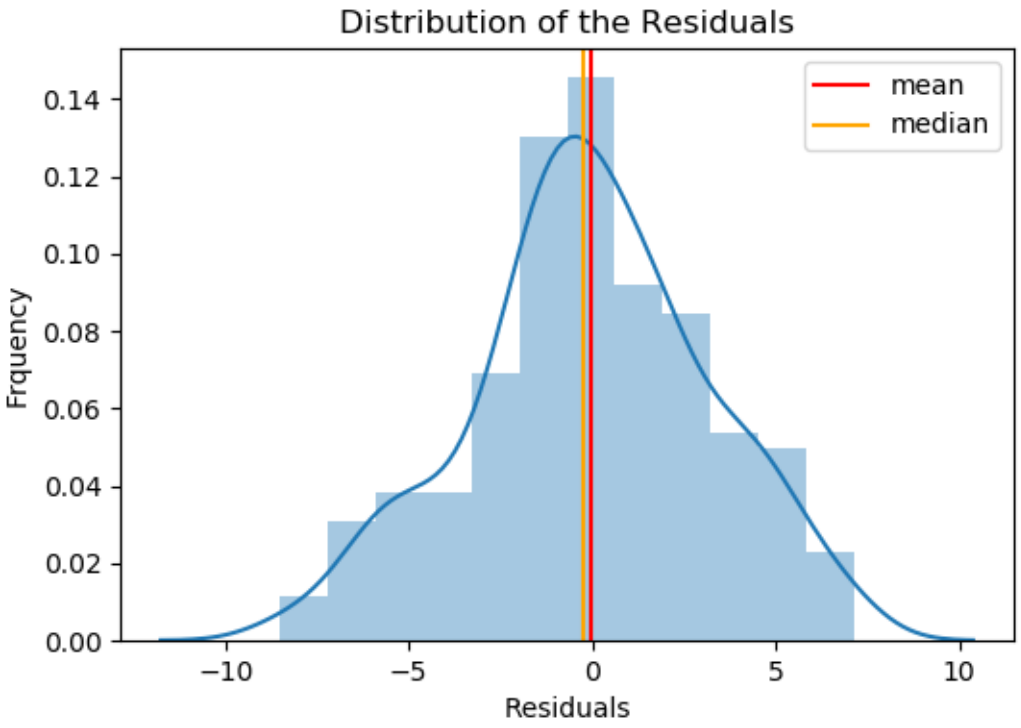
In the perfect model, all the points should be on a straight line. In general, the predictions follow the actual sales. That is good.

Step 8: Verify the assumptions

We have made some assumptions for linear regression. Now, it is time to verify them.

Assumption: The residuals (observed values-predicted values) are approximately normally distributed with the mean 0 and a fixed standard deviation — We'll verify this assumption by drawing a histogram of residuals.

```
fig, ax = plt.subplots(figsize=(6, 4))
plt.style.use('default')
residuals = (y - y_pred)

sns.distplot(residuals.flatten(), ax=ax)
ax.axvline(x=np.mean(residuals), color='red', label='mean')
ax.axvline(x=np.median(residuals), color='orange', label='median')
ax.set_xlabel('Residuals')
ax.set_ylabel('Frquency')
ax.set_title('Distribution of the Residuals')
ax.legend(loc='upper right')
plt.savefig('Distribution of the Residuals.png')
```



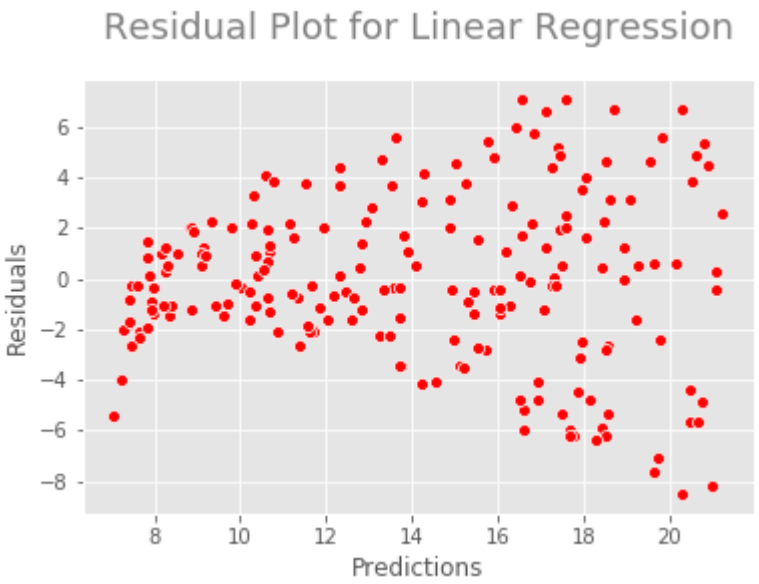
By looking at the histogram, we can verify that the residuals are approximately normally distributed with mean 0. The mean of the residuals is -0.061 which is very close to 0.

```
np.mean(residuals)
```

```
-0.06054000000000055
```

Assumption: The residuals are uncorrelated or independent — We'll verify this assumption by drawing a residual plot.

```
fig, ax = plt.subplots()

residuals = y - y_pred

sns.scatterplot(x=y_pred.flatten(), y=residuals.flatten(), ax=ax, color='red')

ax.set_title('Residual Plot for Linear Regression', pad=20, size=18, color='gray')
ax.set_xlabel('Predictions')
ax.set_ylabel('Residuals')
plt.savefig('residual plot.png')
```



From this plot, we can clearly see some kind of non-linear pattern between predictions and residuals. Ideally, we should not see any pattern in this plot; the presence of the pattern means that we are not using all the information in the features to predict the outcome. This means we have room for improvement.

Approach 2: Linear Regression with the Scikit-learn LinearRegression estimator

Here we are using the popular Scikit-learn Machine Learning library. So, we can train the model very easily. The algorithm using here is **Singular-Value Decomposition (SVD)**.

Step 1: Define X and y

```
X = df[['TV']].values # feature matrix
y = df['sales'].values # target vector

print('X.shape: ', X.shape)
print('y.shape: ', y.shape)
```

```
X.shape:  (200, 1)
y.shape:  (200,)
```

Step 2: Split the dataset into a training set and a testing set

Instead of training the model using all of the rows in the dataset, I'm going to split it into two sets, one for training and one for testing. To do so, I use the **train_test_split()** function (which is in the **model_selection** submodule in **Scikit-learn** library). This function allows you to split the dataset into random train and test subsets called **X_train**, **X_test**, **y_train** and **y_test**. The following code snippet splits the dataset into a 75% training and 25% testing set. The **random_state** parameter of the **train_test_split()** function specifies the seed used by the random number generator. If this is not specified, every time you run this function you will get a different training and testing set. To get the same training and testing sets across different executions, you can specify this parameter with any integer. The popular integers are 0, 1 and 42.

```
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=1)
```

```
print('X_train: ', X_train.shape)
print('X_test: ', X_test.shape)
print('y_train: ', y_train.shape)
print('y_test', y_test.shape)
```

```
X_train:  (150, 1)
X_test:  (50, 1)
y_train:  (150,)
y_test (50,)
```

Step 3: Create and fit (train) the model

Now we can create a simple linear regression model for our data. For this, we can use Scikit-learn **LinearRegression** estimator. You can think a Scikit-learn estimator as a machine learning algorithm. To build a linear regression model using this estimator, all you need to do is to create an instance of **LinearRegression()** class and use **X_train, y_train** to train the model using the **fit()** method of that class.

```
from sklearn.linear_model import LinearRegression

model = LinearRegression()
model
```

```
LinearRegression(copy_X=True, fit_intercept=True, n_jobs=None, normalize=False)
```

Now, the variable **model** is an instance of the **LinearRegression()** class. To train the model, we can use the **fit()** method of that class.

```
model.fit(X_train, y_train)
```

```
LinearRegression(copy_X=True, fit_intercept=True, n_jobs=None, normalize=False)
```

That's it. We have just trained our model. Now, it is time to make some predictions. Before that, it is important to view the **Gradient (Slope)** and **Intercept** of the linear regression line.

Step 4: Get the Gradient and Intercept of the linear regression line

- **Intercept:** We use the **intercept_** attribute.
- **Gradient (Slope):** We use the **coef_** attribute.

```
model.intercept_
```

```
6.91197261886872
```

```
model.coef_
```

```
array([0.04802945])
```

The values are slightly different from the values of the model parameters that we obtained earlier using the gradient descent algorithm.

Step 5: Make predictions

Once we have fitted (trained) the model, we can start to make predictions using the **predict()** method. We pass the values of **X_test** to this method and compare the returned (predicted) values called **y_pred** with **y_test** values.

```
y_pred = model.predict(X_test)
```

```
dic = {'Sales (Actual)': y_test, 'Sales (Predicted)': y_pred}
df1 = pd.DataFrame(dic)
df1.head(10)
```

	Sales (Actual)	Sales (Predicted)
0	23.8	17.036581
1	16.6	16.637936
2	9.5	11.508391
3	14.8	20.369825
4	17.6	19.101847
5	25.5	20.533125
6	16.9	14.755182
7	12.9	15.595697
8	10.5	10.302852
9	17.1	17.257516

It works, although the predictions are not exactly accurate.

To make a prediction for a value that is not in our dataset, we can also use the **predict()** method. We pass the x-value to this method. The x-value should be in the form of a 2d array-like object such as a 2D numpy array, a DataFrame or even a Python lists of lists.

```
model.predict([[30]])
```

```
array([8.35285612])
```

The predicted sales for the TV advertising spending of $30 is approximately 8.35 y units. We can verify the predicted value by using the linear regression formula that was given earlier:

$$sales = \theta_0 + \theta_1 \times TV\,AdSpending$$

$$\theta_0 = 6.912$$
$$\theta_1 = 0.048$$

```
model.intercept_ + (model.coef_ * 30)
```

```
array([8.35285612])
```

The results are exactly the same.

Step 6: Evaluate the model performance

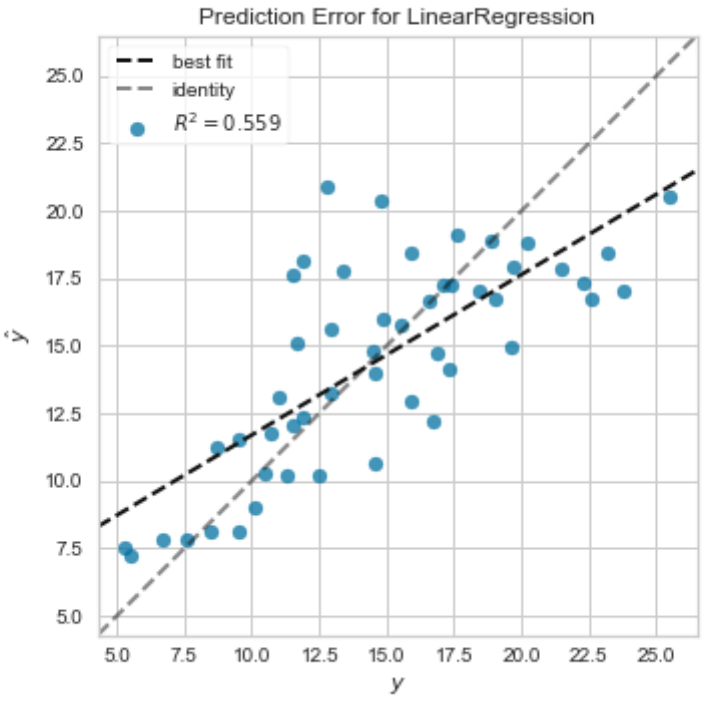
Earlier, we calculated the **Root Mean Squared Error (RMSE)** and the **R-squared (R^2-coefficient of determination)** to evaluate the regression model. Here we also calculate them but with using Scikit-learn **metrics** submodule.

```
from sklearn.metrics import r2_score, mean_squared_error
import numpy as np

R_squared = r2_score(y_test, y_pred)
MSE = mean_squared_error(y_test, y_pred)
RMSE = np.sqrt(MSE)

print('R^2: ', np.round(R_squared, 2))
print('RMSE: ', np.round(RMSE, 2))
```

```
R^2:  0.56
RMSE:  3.21
```

As a graphical method, we can use a prediction error plot to evaluate the regression model. This time, I use the **Yellowbrick** module to make the plot.

```
from yellowbrick.regressor import PredictionError

# Instantiate the visualizer
visualizer_1 = PredictionError(model)
# Fit the training data to the visualizer
visualizer_1.fit(X_train, y_train)
# Evaluate the model on the test data
visualizer_1.score(X_test, y_test)
# Finalize and render the figure
visualizer_1.show(outpath = 'pred error.png')
```

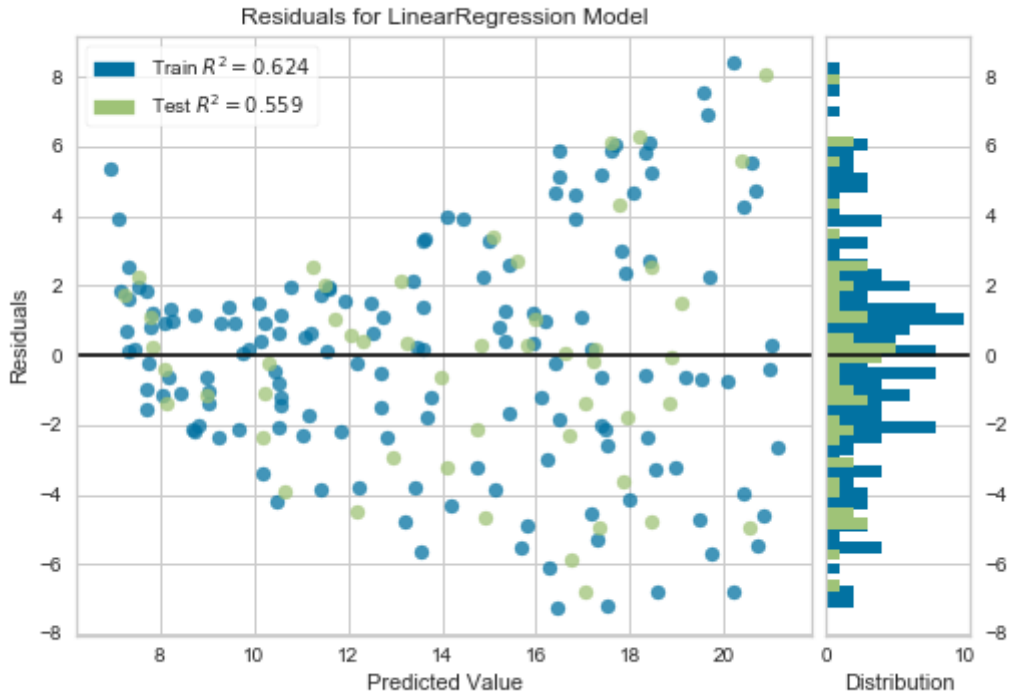


Ideally, the plot should be a straight line but for now, it is good enough. We can compare this plot against the 45-degree line, where the prediction exactly matches the model.

Step 7: Verify the assumptions

To verify the assumptions of our linear regression model, I create the histogram distribution of residuals and the residual plot in the same graph using the **Yellowbrick** module.

```
from yellowbrick.regressor import ResidualsPlot

# Instantiate the visualizer
visualizer_2 = ResidualsPlot(model)
# Fit the training data to the visualizer
visualizer_2.fit(X_train, y_train)
# Evaluate the model on the test data
visualizer_2.score(X_test, y_test)
# Finalize and render the figure
visualizer_2.show(outpath = 'residual plot.png')
```



We can verify that the residuals are approximately normally distributed. But we can clearly see some kind of non-linear pattern between predictions and residuals.

Approach 3: Linear Regression with the Normal Equation The Normal Equation

To find the value of $\boldsymbol{\theta}$ that minimizes the cost function, there is a mathematical equation that gives the result directly. This is called the **Normal Equation**.

$$\theta = (X^T X)^{-1} X^T y$$

In this equation:

- $\boldsymbol{\theta}$**:** Parameter matrix which contains optimized values. It is returned as a 2d numpy array. The shape is **(n+1, 1)** where **n** is the number of predictors.
- **X:** Feature matrix. It is represented as a 2d numpy array. The shape is **(m, n+1)** where **n** is the number of predictors and **m** is is the number of training examples (rows/observations). The elements of the first column are all set to ones. This is to accommodate the intercept term.

- **y:** Target vector. It is represented as a 2d numpy array to match the dimension. The shape is **(m, 1)** where **m** is the number of training examples (rows/observations).

Computational Complexity

The Normal Equation computes the inverse of **X.transpose() * X**, which is an **(n + 1) × (n + 1)** matrix (where **n** is the number of predictors). With the normal equation, computing the inversion has complexity $O(n^3)$. So if we have a very large number of predictors, the normal equation will be slow. In practice, when n exceeds 10,000 it might be a good time to go from a normal solution to the gradient descent.

The Singular Value Decomposition (SVD) approach used by Scikit-Learn's LinearRegression class is about $O(n^2)$. If you double the number of predictors, you multiply the computation time by roughly 4. The gradient descent approach is about $O(kn^2)$. If you double the number of predictors, you multiply the computation time by roughly 4*k.

Both the Normal Equation and the SVD approach get very slow when the number of features grows large (e.g., 10,000). On the positive side, both are linear with regard to the number of instances in the training set (they are O(m)), so they handle large training sets efficiently, provided they can fit in memory.

Calculate optimized θ values using the normal equation

```
import numpy as np
import pandas as pd

df = pd.read_csv('Advertising.csv')
df.drop(columns=['radio', 'newspaper'], inplace=True)
m = df['sales'].values.size
X = np.append(np.ones((m, 1)), df['TV'].values.reshape(m, 1), axis=1)
y = df['sales'].values.reshape(m, 1)
A = np.linalg.inv(np.dot(X.transpose(), X))
B = np.dot(X.transpose(), y)
theta = np.dot(A, B)
theta
```

```
array([[7.03259355],
       [0.04753664]])
```

$\theta_0 = 7.033$
$\theta_1 = 0.048$

$$sales = 7.033 + 0.048 \times TV\ AdSpending$$

The optimized parameter values are very similar to the values obtained in approach 1 — Linear Regression with Gradient Descent.

Summary

Here is the comparison between the gradient descent and the normal equation.

Gradient Descent	Normal Equation
Need to choose alpha	No need to choose alpha
Needs many iterations	No need to iterate
O (kn^2)	O (n^3), need to calculate inverse of $X^T X$
Works well when n is large	Slow if n is very large

What is next?

With R^2=0.61, our model is not actually very good. From the residual plot, we have seen some kind of non-linear pattern between predictions and residuals. All these things suggest that we are not using all the information in the predictors to predict the outcome. This means we have room for improvement of the model by including multiple predictors.

Earlier I said, "The predictive analytics process is an iterative process in which you will always find yourself going back and forth between the steps to build a better model which captures a great amount of variability in your data". I'll go to step 1 and change the problem definition to "Build a Multiple Linear Regression Model to predict sales based on all important predictors". Then I will follow other steps to finalize the model.

So, in the next article, I'll discuss how to create a multiple linear regression model to predict sales by including all important predictors in the Advertising dataset. I'll also discuss feature selection techniques there. Goodbye for now!

Data Science 365

Bring data into actionable insights.

Data Science 365

This tutorial was designed and created by Rukshan Pramoditha, the Author of Data Science 365 Blog.

Technologies used in this tutorial

- **Python** (high-level programming language)
- **Numpy** (numerical Python library)
- **pandas** (Python data analysis and manipulation library)
- **matplotlib** (Python data visualization library)
- **seaborn** (Python advanced data visualization library)
- **Yellowbrick** (Machine learning visualization library)
- **Scikit-learn** (Python machine learning library)
- **Jupyter Notebook** (Integrated Development Environment)

Mathematics used in this tutorial

- **Linear algebra**
- **Partial derivative**

Machine learning used in this tutorial

- **Linear regression algorithm**
- **Gradient descent algorithm**

Project management used in this tutorial

- **The concept of Work Breakdown Structure (WBS)**

Analytics used in this tutorial

- **Predictive analytics process**

2020–06–14