Google Play Store Apps Prediction

Project Report Submitted for

Statistical Learning Theory Course Project

Submitted by-

Pratyush Parashar (201020441)

Vikalp Kumar Tripathi (201020258)

Dr. Shyama Prasad Mukherjee

International Institute of Information Technology, Naya Raipur

(A Joint Initiative of Govt. of Chhattisgarh and NTPC)

1. INTRODUCTION

1.1. Regression Analysis

Regression Analysis is a proven method for determining which variables have an impact on a given topic. Regression analysis allows you to confidently determine which factors are most important, which can be ignored, and how these factors interact. The goal of regression analysis is to use one or more independent or control variables to explain variability in the dependent variable.

Types of Regression Analysis Algorithms include-

- **Linear Regression**
- **Multiple Linear Regression**
- **Polynomial Regression**
- **Gradient Descent Method**
- **Regularization**
    - **Ridge Regression**
    - **Lasso Regression**

1.2. Description of Dataset

Google Play Store Apps Dataset gives us the detail of the different apps in the Play Store and their correlated features i.e., Rating, Installs, Customer Reviews, Android Version, App Size, etc. These are the features using which we can predict whether the app is suitable for a particular user or not. We can also visualize the Genres of Apps that are performing well in the market at a given point of time. For Example, Social Media Apps are in much more demand nowadays than they were in the past. The Dataset consists of 13 different attributes or features of data regarding 10841 different Apps uploaded on the Google Play Store.

2. LITERATURE

2.1. Linear Regression

Linear Regression is a model that uses a straight line to estimate the relationship between one independent variable and one dependent variable, with both variables being quantitative. Linear Regression Analysis is a statistical technique for predicting the value of one variable based on the value of another. The dependent variable is the variable you want to predict. Linear regression creates a straight line or surface that reduces the difference between predicted and actual output values.

2.2. Multiple Linear Regression

Multiple Linear Regression is a model that uses a straight line to estimate the relationship between multiple independent variables and one dependent variable, with both variables being quantitative. Multiple Linear Regression creates a straight line or surface that reduces the differences between predicted and actual output values. It's a statistical method for calculating the value of a criterion based on the values of several other independent variables. It is the simultaneous assessment of multiple factors in order to determine how and to what extent they influence a specific outcome.

Difference b/w Linear Regression and Multiple Linear Regression

By determining the slope and intercept that define the line and minimize regression errors, linear regression attempts to draw a line that comes closest to the data. A multiple linear regression is when two or more explanatory variables have a linear relationship with the dependent variable.

2.3. Polynomial Regression

Polynomial Regression is a regression algorithm that uses an nth degree polynomial to model the relationship between a dependent(y) and independent variable(x). Polynomial Regression is a type of linear regression that estimates the relationship as an nth degree polynomial. It is a special case of Multiple Linear Regression. Because Polynomial Regression is sensitive to outliers, the presence of one or two of them can have a negative impact on the results. When the points in the data are not captured by the Linear Regression Model and the Linear Regression fails to clearly describe the best result, Polynomial Regression is used.

2.4. Gradient Descent Method

Gradient Descent is a first-order iterative algorithm for solving optimization problems. It's an iterative optimization algorithm for finding a function's local minimum. To use gradient descent to find a function's local minimum, we must take steps proportional to the negative of the function's gradient (move away from the gradient) at the current point.

2.5. Regularization

Regularization is a technique for improving the generalization of a learning algorithm by making minor changes to it. As a result, the model's performance on previously unseen data improves as well. One of the most important concepts in machine learning is regularization. It aids in the prevention of overfitting, increases model robustness, and reduces model complexity. Regularization is the process of shrinking or regularizing the coefficients towards zero in machine learning.

2.5.1. Ridge Regression

Ridge Regression is a model tuning technique that can be used to analyze data with multicollinearity. L2 regularization is achieved using this method. When there is a problem with multicollinearity, least-squares are unbiased, and variances are large, resulting in predicted values that are far from the actual values. When the number of predictor variables in a set exceeds the number of observations, or when a data set has multicollinearity, it is a way to create a parsimonious model (correlations between predictor variables).

2.5.2. Lasso Regression

Lasso (least absolute shrinkage and selection operator) is a regression analysis method in statistics and machine learning that performs both variable selection and regularization in order to improve the predictability and interpretability of the resulting statistical model. It is a statistical formula used to select features for data models and to regularize them.

3. METHODOLOGY

3.1. Data Processing

The task of converting data from one form to a much more usable and desired form, i.e. making it more meaningful and informative, is known as data processing. This entire process can be automated using Machine Learning algorithms, mathematical modelling, and statistical knowledge. We have made use of Techniques of Data Processing for the completion of the Data Processing Course Project in the previous half of the semester.

We applied following techniques for further applying Regression Algorithms such as-

- **Data Imputation**
- **Categorical Encoding**
- **Discretization**

3.2. Architecture

Regression Analysis is a proven method for determining which variables have an impact on a given topic. Regression Analysis allows us to confidently determine which factors are most important, which factors can be ignored, and how these factors interact.

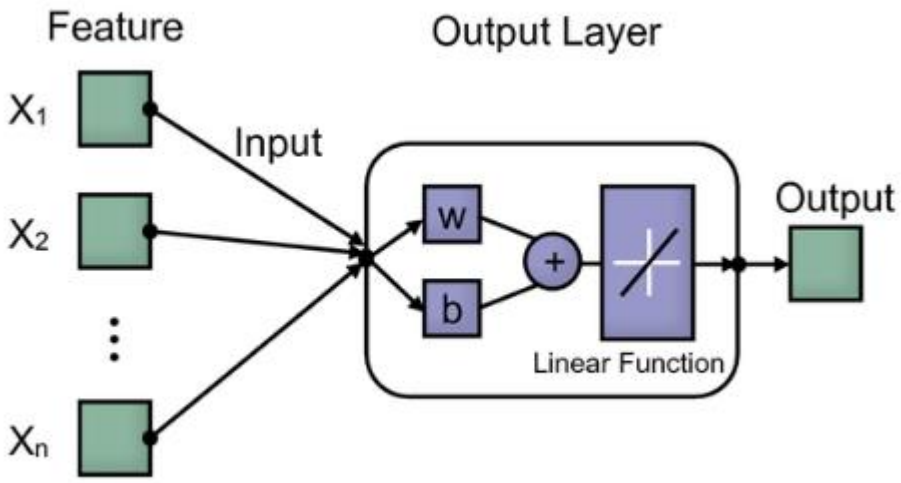
To fully comprehend regression analysis, you must first understand the following terms:

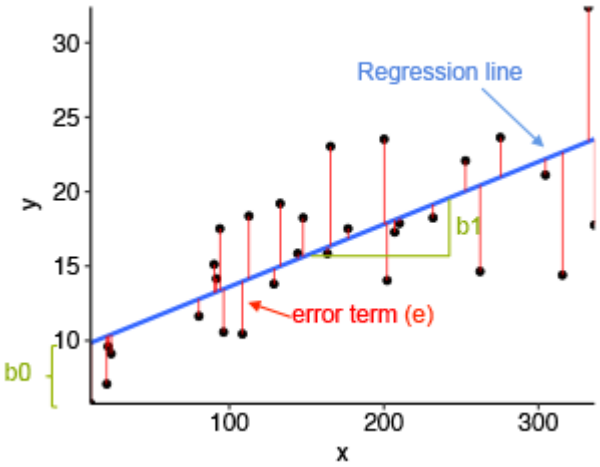
Dependent Variable: This is the main factor that we are trying to understand or predict.

Independent Variables: These are the factors that we hypothesize have an impact on our dependent variable.

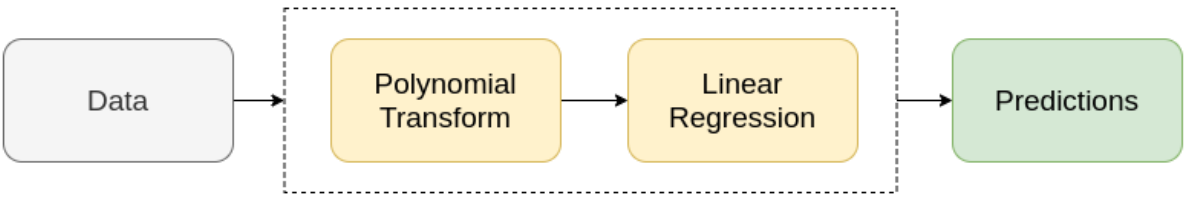
In order to conduct a regression analysis, we will need to define a dependent variable that you hypothesize is being influenced by one or several independent variables. We have further demonstrated regression analysis using all the different methods discussed above, the results of which we will be pasting in the Results Section. We can then visualize the predictions done using the output of regression analysis on our dataset.

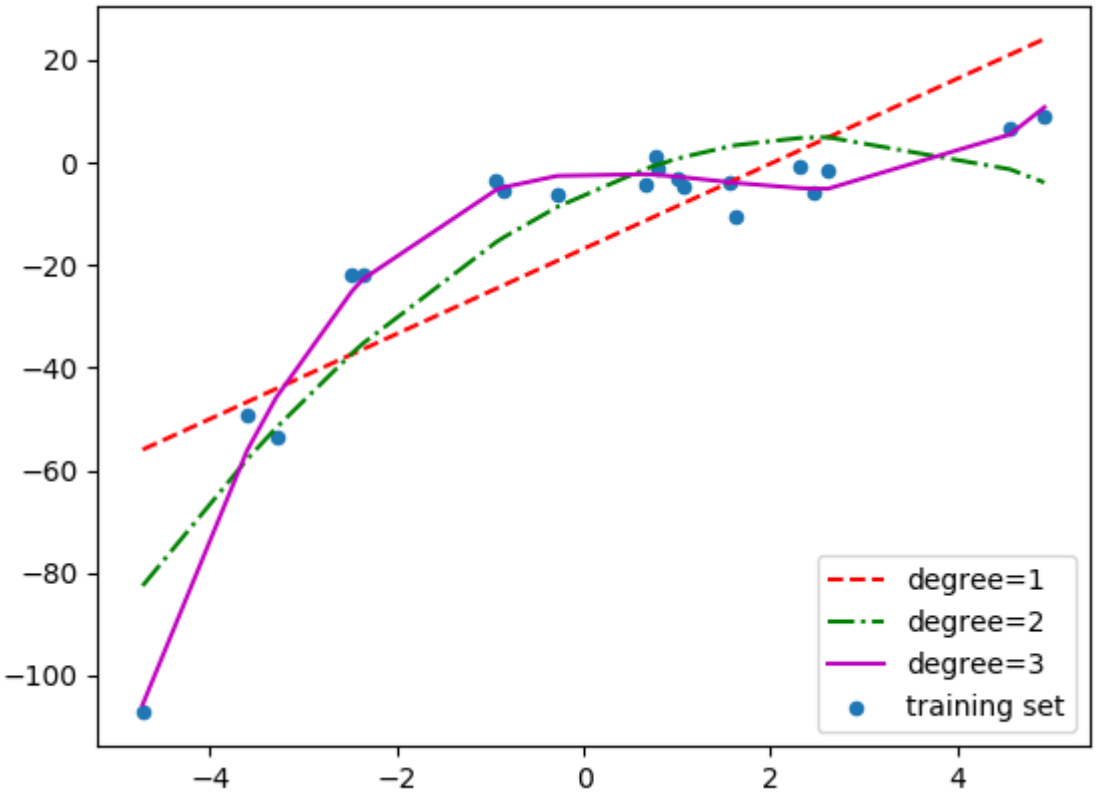
Multiple Linear Regression



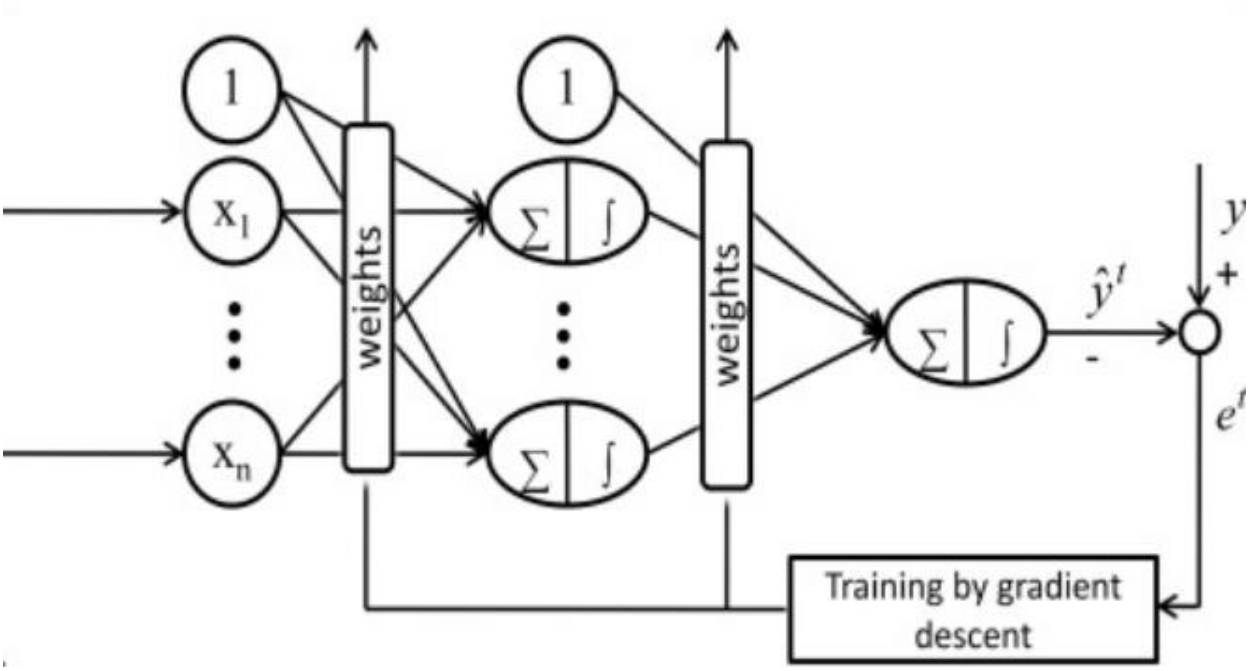


Polynomial Regression

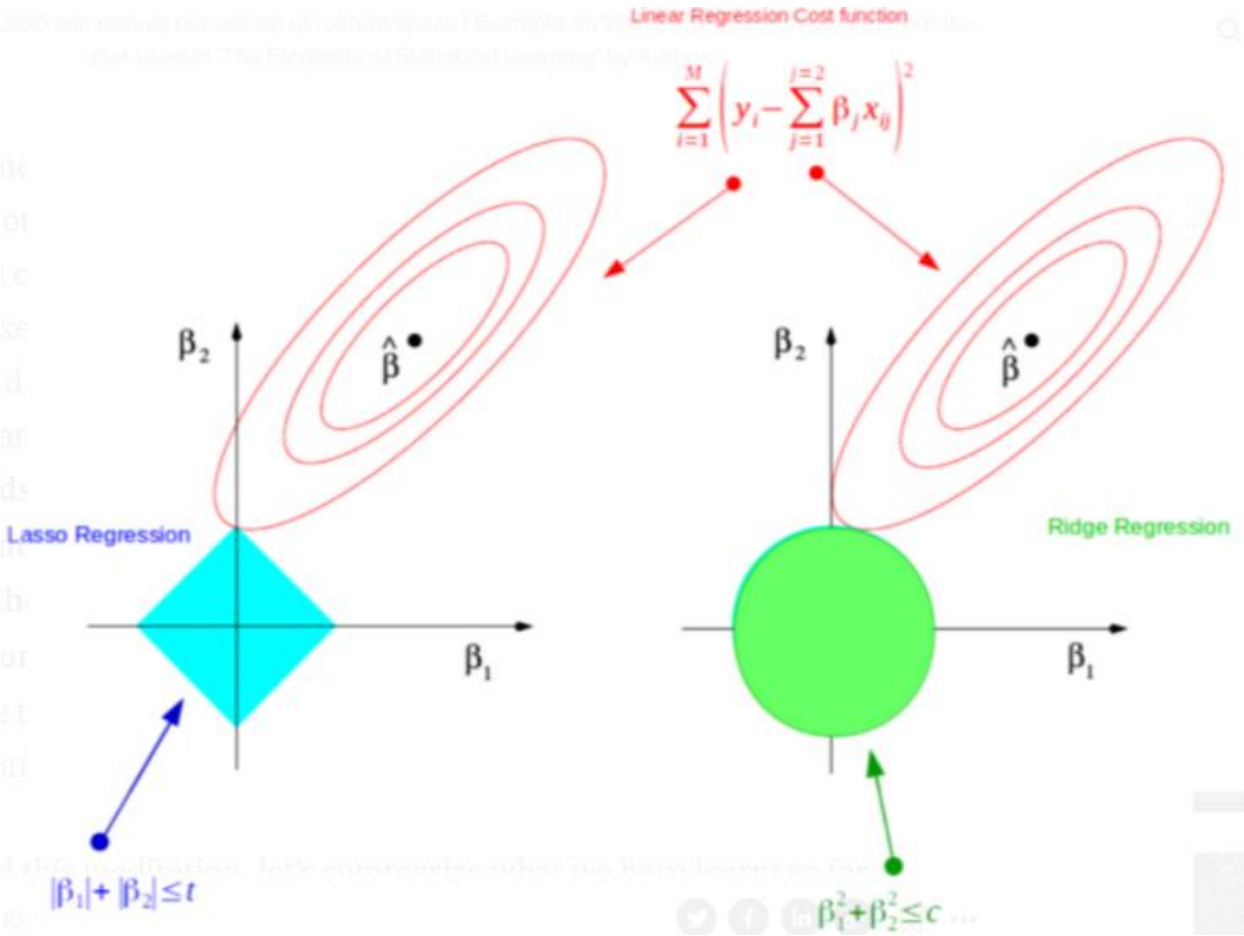




Gradient Descent



Ridge and Lasso Regression



3.3. Techniques

3.3.1. Linear Regression

Assumptions:

1. Linearity: The dependent variable Y should have a linear relationship with the independent variables. A scatter plot between both variables will be used to test this assertion.

2. Normality: The X and Y variables should be distributed regularly. The Normality assumption can be tested using histograms, KDE plots, and Q-Q graphs.

3. Homoscedasticity: For all values of X, the variance of the error terms should be constant, i.e., the spread of residuals should be constant.

4. Independence/No Multicollinearity: The variables should be independent of each other i.e.; no correlation should be there between the independent variables. To check the assumption, we can use a correlation matrix or VIF score.

Algorithm:

1. Data Preprocessing

2. Fitting the Simple Linear Regression to the Training Set

3. Prediction of Test Set Result

4. Visualizing the Training Set Results

5. Visualizing the Testing Set Results

Steps:

1: We have cleaned the data with the help of various preprocessing techniques such as imputation, categorical encoding, discretization etc.

2: We have identified ‘Rating’ as our dependent variable which we need to predict. With the help of correlation matrix, we have also identified its correlation with other features.

3: Identified that ‘Rating’ is strongly correlated with ‘Reviews’, ‘Installs’ and ‘Size’ so performed linear regression on each of them for 70:30 train-test split.

4: Plotted the scatter plot, the best fit line and then calculated different kind of errors for training as well as testing data.

3.3.2. Multiple Linear Regression

Assumptions:

1. The result variable and the independent factors must have a linear relationship.

2. Multivariate Normality i.e., this algorithm assumes that the residuals are normally distributed.

3. Multiple Linear Regression is based on the assumption that the independent variables are not substantially connected.

4. The variance of error terms is similar across independent variable values.

5. At least two independent variables, which might be nominal, ordinal, or interval/ratio level variables, are required for multiple linear regression.

Algorithm:

1. Data Preprocessing Steps

2. Identify dependent and independent attribute or features from the dataset.

3. Fitting our MLR Model to the Training set

4. Prediction of Test Set Results

5. Visualize the Training and Test Set Results.

Steps:

1: We have cleaned the data with the help of various preprocessing techniques such as imputation, categorical encoding, discretization etc.

2: We have identified ‘Rating’ as our dependent variable which we need to predict. With the help of correlation matrix, we have also identified its correlation with other features.

3: Identified that ‘Rating’ is strongly correlated with ‘Reviews’, ‘Installs’ and ‘Size’ so performed linear regression on each of them for 70:30 train-test split.

4: Plotted the scatter plot, calculated the difference between actual and predicted values (which came out to be very little) and then used it to find various kind of other errors.

5: Checked this model accuracy with other models like KNN, Random Forest etc. and came out

to conclusion that for this combination, **Random Forest Regressor will be best.**

3.3.3. Polynomial Regression

Assumptions:

1) The behavior of a dependent variable y can be explained by a linear, curvilinear or additive relationship between the dependent variable and a set of k independent variables (xi, i=1 to k)).

2) There is a linear or curvilinear relationship between the dependent variable y and any independent variable xi.

3) The independent variables x_i are independent of each other.

4) The errors are independent, normally distributed with mean zero and a constant variance.

Algorithm:

- Data Preprocessing
- Build a Linear Regression Model and fit it to the dataset
- Build a Polynomial Regression Model and fit it to the dataset
- Visualize the result for Linear Regression and Polynomial Regression Model.
- Predicting the output.

Steps:

1) We have cleaned the data with the help of various preprocessing techniques such as imputation, categorical encoding, discretization etc.

2) Identified independent and dependent features. Built linear regression model.

3) After transforming the train data with the help of Polynomial Feature, we worked on Polynomial Regression model for degree values of 1 to 5.

4) Visualized the output and calculated error values.

5) Made a summary with the help of OLS which includes the value of AIC, BIC, CP etc.

3.3.4. Gradient Descent Method

Assumption:

Gradient Descent is based on the assumption or observation that if a multi-variable function F(x) is defined and differentiable in the vicinity of point a, then F(x) drops the fastest if one moves from a in the direction of F's negative gradient (x).

Algorithm:

Gradient Descent Algorithm's purpose is to minimize a given function (say cost function). It executes two phases iteratively to attain this goal:

1) Calculates the function's gradient (slope) and first order derivative at that point.

2) Makes a step (move) in the opposite direction of the gradient, increasing the slope by alpha times the gradient at that point from the current position.

Steps:

We have implemented Gradient Descent for each combination of strong features, for various learning rates (0.1,0.001,0.5,0.05,1) and for various number of iterations (100,500,1000) and compared the result below by finding error values for each case.

3.3.5. Regularization

3.3.5.1. Ridge Regression

Assumption:

Ridge Regression is based on the same assumptions as linear regression: linearity, constant variance, and independence. As ridge regression does not provide confidence limits, it is not necessary to assume that the error distribution is normal.

Algorithm:

1) Data Preprocessing

2) Train-Test Split

3) Invoke the Linear Regression function and find the Best Fit Model on Training Data

4) Explore the coefficients for each of the independent attributes

5) Plot and visualize the data using coefficients.

Implementation:

We have implemented Ridge Regression on 3 strongly correlated features Installs, Reviews and Size where we have used alpha =0.05. Here, α (alpha) is the parameter which balances the amount of emphasis given to minimizing RSS vs minimizing sum of square of coefficients. While we do this, there is significant change in value of CP, AIC, BIC as we split the data in different splits whereas there is not much deviation in values for same feature in a particular split

3.3.5.2. Lasso Regression

Assumption:

Ridge Regression is based on the same assumptions as linear regression: linearity, constant variance, and independence. As ridge regression does not provide confidence limits, it is not necessary to assume that the error distribution is normal. Ridge is preferred when there are not much predictors whereas we use Lasso mostly whenever there are many predictors.

Algorithm:

Lasso Regression algorithm main purpose is to find the subset of predictors that produces the least amount of prediction error for a quantitative response variable. The lasso accomplishes this by imposing a constraint on the model parameters that leads some regression coefficients to decrease toward zero. After the shrinking procedure, variables having a regression coefficient of zero are removed from the model. The response variable is most closely connected with variables with non-zero regression coefficients.

Implementation:

We have implemented Lasso Regression on 3 strongly correlated features Installs, Reviews and Size where we have used alpha =0.05. Here, α (alpha) is the parameter which balances the amount of emphasis given to minimizing RSS vs minimizing sum of square of coefficients. While we do this, there is significant change in value of CP, AIC, BIC as we split the data in different splits whereas there is not much deviation in values for same feature in a particular split

4. EXPERIMENTATION AND RESULTS

4.1. Linear Regression

- **Rating vs Reviews**

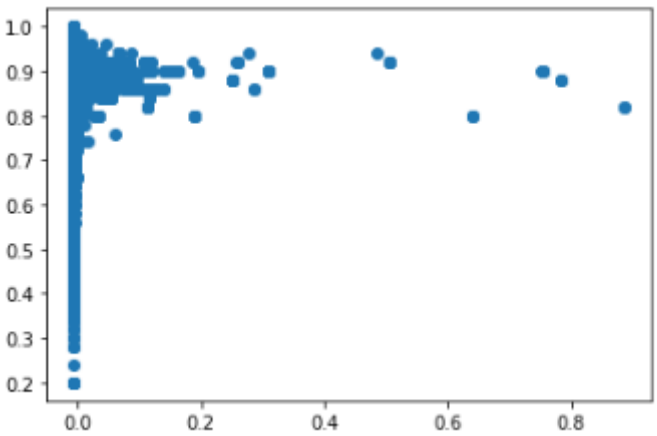


Fig: Scatter plot of Rating and Reviews

```
m,c=np.random.random(),np.random.random()
```

```
def learn(x,y,m,c,epoch):
    for i in range(epoch):
        error=y-(m*x+c)
        x_error=(y-(m*x+c))*x
        error=np.sum(error)/len(x)
        x_error=np.sum(x_error)/len(x)
        learning_rate = 0.01
        delta_m= learning_rate *x_error
        delta_c =learning_rate*error
        m+=delta_m
        c+=delta_c
    return(m,c)
```

```
m,c=learn(x,y,m,c,20000)
```

```
x1=[0,1]
y1=[0*m+c,1*m+c]
```

Fig: Mathematical Approach of Linear Regression

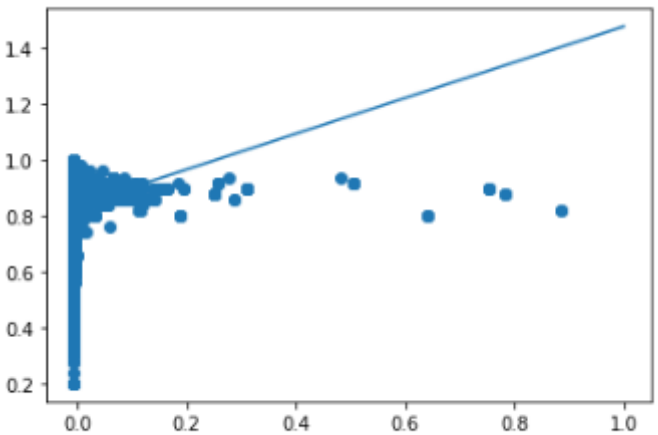


Fig: Best Fit Line plot

Here we have plotted the best fit line with the mathematical approach of Linear Regression where we have taken the learning rate to be 0.01 and iterations value to be 20,000. With the best fit line, we can clearly see that it is increasing and has covered most of the bulk data so it has major role to predict the Rating of any particular app.

```
from sklearn.metrics import mean_squared_error, r2_score, mean_absolute_error
from math import sqrt
print("Coeff: %0.3f" % regressor.coef_)
print("Intercept: %0.3f" % regressor.intercept_)
print("R2 score : %.3f" % r2_score(y_test, regressor.predict(x_test)))
print("MSE: %.3f" % mean_squared_error(y_test, regressor.predict(x_test)))
print("RMSE: %.3f" % sqrt(mean_squared_error(y_test, regressor.predict(x_test))))
print("MAE: %.3f" % sqrt(mean_absolute_error(y_test, regressor.predict(x_test))))
```

```
Coeff: 0.037
Intercept: 4.191
R2 score : 0.003
MSE: 0.229
RMSE: 0.479
MAE: 0.560
```

Fig: Error Calculation

It can be clearly seen that error value of MAE and MSE are indicating that it is a good model. Coefficient as well as intercept value is also clearly visible from the code.

```
#AIC,BIC,RSS
y = train_data['Rating']

#define predictor variables
x = train_data[['Reviews']]

#add constant to predictor vari
x = sm.add_constant(x)

#fit regression model
model = sm.OLS(y, x).fit()

#view AIC of model
print(model.aic)
print(model.bic)
print(model.ssr)
```

```
14764.413645446773
14778.995457486404
2477.032016599371
```

```
model = LinearRegression()
X, y = train_data[["Reviews"]], train_data.R
model.fit(X, y)

#display adjusted R-squared
1 - (1-model.score(X, y))*(len(y)-1)/(len(y
```

```
0.004611067021882742
```

```
#cp

m=len(Y)
p=1
hat_sigma_squared=(1/(m-p-1))*2477.032016599
Cp=(1/m)*(2477.032016599371+2* hat_sigma_squ
Cp
```

```
0.22857174647959502
```

Fig: AIC, BIC, RSS, CP and Adjusted R square value is calculated

Here we have calculated all these values with the help of OLS (ordinary least square) and with the mathematical approach for cp and adjusted r square.

- **Rating vs Installs**

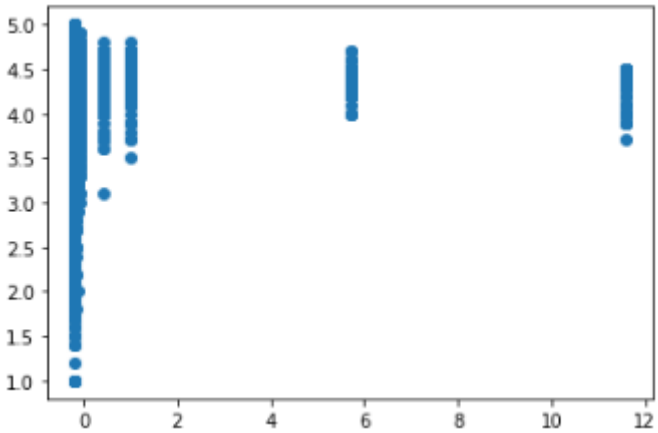


Fig :Scatter plot of Rating and Installs

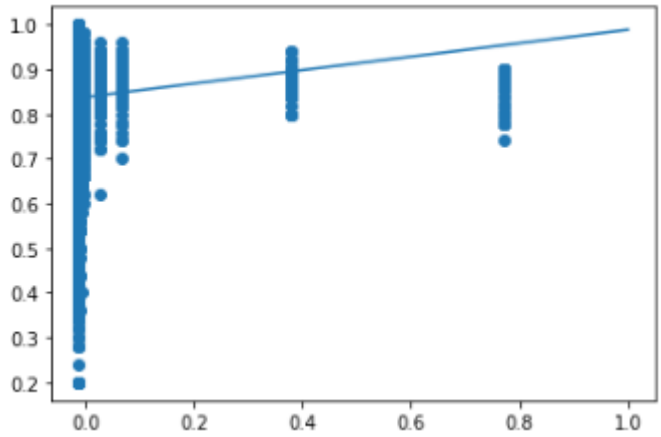


Fig: Best Fit Line plot

Here we have plotted the best fit line with the mathematical approach of Linear Regression where we have taken the learning rate to be 0.01 and iterations value to be 20,000. With the best fit line, we can clearly see that it is increasing and has covered most of the bulk data so it has major role to predict the Rating of any particular app.

```
from sklearn.metrics import mean_squared_error, r2_score, mean_absolute_error
from math import sqrt
print("Coeff: %0.3f" % regressor.coef_)
print("Intercept: %0.3f" % regressor.intercept_)
print("R2 score : %.3f" % r2_score(y_test, regressor.predict(x_test)))
print("MSE: %.3f" % mean_squared_error(y_test, regressor.predict(x_test)))
print("RMSE: %.3f" % sqrt(mean_squared_error(y_test, regressor.predict(x_test))))
print("MAE: %.3f" % sqrt(mean_absolute_error(y_test, regressor.predict(x_test))))
```

```
Coeff: 0.026
Intercept: 4.191
R2 score : 0.002
MSE: 0.229
RMSE: 0.479
MAE: 0.561
```

Fig: Error Calculation

```
#AIC,BIC,RSS
y = train_data['Rating']

#define predictor variables
x = train_data[['Installs']]

#add constant to predictor variab
x = sm.add_constant(x)

#fit regression model
model = sm.OLS(y, x).fit()

#view AIC of model
print(model.aic)
print(model.bic)
print(model.ssr)
```

```
14786.263645226072
14800.845457265703
2482.0304240419223
```

```
model = LinearRegression()
X, y = train_data[["Reviews"]], train_data.Rating
model.fit(X, y)

#display adjusted R-squared
1 - (1-model.score(X, y))*(len(y)-1)/(len(y)-X.shape
```

```
0.004611067021882742
```

```
#cp

m=len(Y)
p=1
hat_sigma_squared=(1/(m-p-1))*2482.0304240419223
Cp=(1/m)*(2482.0304240419223+2* hat_sigma_squared)
Cp
```

```
0.22903298182540577
```

Fig: AIC, BIC, RSS, CP and Adjusted R square value is calculated

In the above figures we have calculated of MAE and MSE. They are clearly indicating that it is a good model. Coefficient as well as intercept value is also clearly visible from the code. Here we have also calculated AIC, BIC, RSS and all these values are calculated with the help of OLS (ordinary least square) whereas with the mathematical approach for cp and adjusted r square.

- **Rating vs App Size**

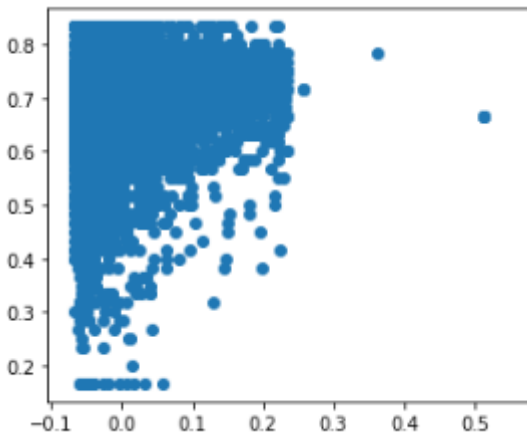


Fig: Scatter plot of Rating and Size

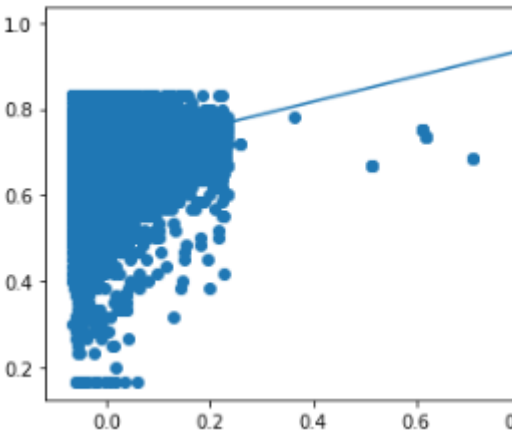


Fig: Best Fit Line plot

Here we have plotted the best fit line with the mathematical approach of Linear Regression where we have taken the learning rate to be 0.01 and iterations value to be 20,000. With the best fit line, we can clearly see that it is increasing and has covered most of the bulk data so it has major role to predict the Rating of any particular app. We have also calculated all kind of errors and AIC, BIC values with the help of code which we have seen above.

4.2. Multiple Linear Regression

Here we will take together Reviews, Installs and Size and we will plot the model and find the difference between actual and predicted values. Here, we will also use same predictors for KNN, Random Forest and Bagging Regressor to find which one of them is best.

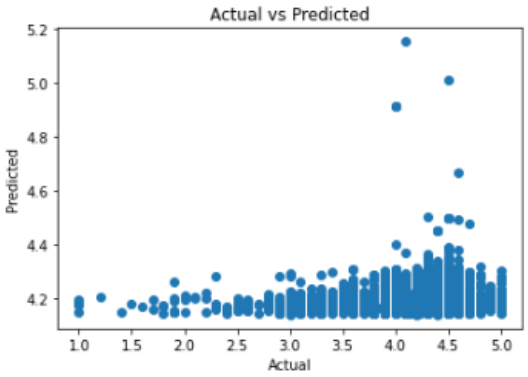


Fig: Plot of Actual and Predicted Data

As it can be visible from the plot that there is very less difference between the actual and predicted value which can also be visualized from the below figure in which we have shown the exact difference of first 20 values.

```
pred_y_df=pd.DataFrame({'Actual Value':y_test,'Predicted value':y_pred,'Difference': y_test-y_pred})
pred_y_df[0:20]
```

	Actual Value	Predicted value	Difference
0	4.300000	4.315727	-0.015727
1	3.600000	4.185045	-0.585045
2	4.500000	4.179191	0.320809
3	4.400000	4.208801	0.191199
4	4.400000	4.159018	0.240982
5	4.500000	4.250957	0.249043
6	4.400000	4.201490	0.198510
7	4.238189	4.242144	-0.003955
8	4.600000	4.222527	0.377473
9	3.800000	4.212885	-0.412885
10	3.600000	4.173802	-0.573802
11	4.600000	4.196759	0.403241
12	4.183097	4.192219	-0.009122
13	4.700000	4.204908	0.495092
14	4.000000	4.238316	-0.238316
15	4.300000	4.209953	0.090047
16	4.000000	4.279267	-0.279267
17	4.200000	4.147493	0.052507

Fig: Table showing difference between actual and Predicted Data

After applying this regression technique, we have also calculated the MSE and MAE value for test data whose values are 0.225 and 0.310 respectively.

KNN Model: K-Nearest Neighbors Model

```
from sklearn.neighbors import KNeighborsRegressor
```

```
# Fit model
knr = KNeighborsRegressor(n_neighbors = 5)
knr.fit(X_train, y_train)

# Measure mean squared error for training and validation sets
print('Mean squared Error for Training Set:', mean_squared_error(y_train, knr.predict(X_train)))
print('Mean squared Error for Test Set:', mean_squared_error(y_test, knr.predict(X_test)))
```

```
Mean squared Error for Training Set: 0.1618615339949217
Mean squared Error for Test Set: 0.24331168271222978
```

Fig: Applying KNN for Prediction

Here the MSE for train and test set are 0.16 and 0.24 respectively for neighbors value equal to 5.

Random Forest Regressor

```
from sklearn.ensemble import RandomForestRegressor
```

```
# Train Random Forest Regressor
randomf = RandomForestRegressor(n_estimators=300)
randomf.fit(X_train, y_train)

# Measure mean squared error for training and validation sets
print('Mean squared Error for Training Set:', mean_squared_error(y_train, randomf.predict(X_train)))
print('Mean squared Error for Test Set:', mean_squared_error(y_test, randomf.predict(X_test)))
```

```
Mean squared Error for Training Set: 0.028226766895962947
Mean squared Error for Test Set: 0.19902366809134187
```

Fig: Applying Random Forest Regressor for prediction

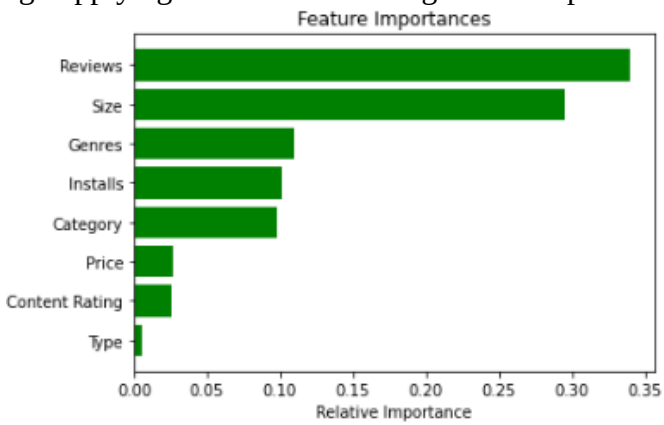


Fig: Feature importance in Random Forest

It is clearly visible that if we apply random forest regressor then the most important feature is Reviews and then Size.

Bagging Regressor

```
from sklearn.ensemble import BaggingRegressor
```

```
# Fit model
br = BaggingRegressor(random_state=300)

br.fit(X_train, y_train)

# Measure mean squared error for training and validation sets
print('Mean squared Error for Training Set:', mean_squared_error(y_train, br.predict(X_train)))
print('Mean squared Error for Test Set:', mean_squared_error(y_test, br.predict(X_test)))
```

```
Mean squared Error for Training Set: 0.03989299884581395
Mean squared Error for Test Set: 0.21711816904674575
```

Fig: Applying Bagging Regressor for Prediction

So, after applying all these we came to conclusion that from above all techniques, Random Forest Regressor performed best with testing mean squared error of 0.19 which is also confirmed with the help of Lazy-Library which tells us which model is best among all types and it can be shown from below figure.

Model	Adjusted R-Squared	R-Squared	RMSE	Time Taken
GradientBoostingRegressor	0.07	0.07	0.50	0.91
LGBMRegressor	0.06	0.06	0.50	0.17
HistGradientBoostingRegressor	0.05	0.06	0.50	0.48
RandomForestRegressor	0.03	0.04	0.51	3.02
XGBRegressor	0.01	0.01	0.51	0.71
ExtraTreesRegressor	0.00	0.01	0.51	2.02
MLPRegressor	-0.00	0.00	0.51	6.37
GammaRegressor	-0.03	-0.02	0.52	0.03
PoissonRegressor	-0.03	-0.02	0.52	0.04
GeneralizedLinearRegressor	-0.03	-0.02	0.52	0.04
TweedieRegressor	-0.03	-0.02	0.52	0.03
AdaBoostRegressor	-0.03	-0.02	0.52	0.15
BayesianRidge	-0.03	-0.02	0.52	0.05
LassoLarsIC	-0.03	-0.02	0.52	0.03
ElasticNetCV	-0.03	-0.02	0.52	0.39
LassoCV	-0.03	-0.02	0.52	0.19
RidgeCV	-0.03	-0.02	0.52	0.04
Ridge	-0.03	-0.02	0.52	0.02
LinearRegression	-0.03	-0.02	0.52	0.05
TransformedTargetRegressor	-0.03	-0.02	0.52	0.03

Fig: Result Obtained After applying Lazy Library

4.3. Polynomial Regression

- For Reviews

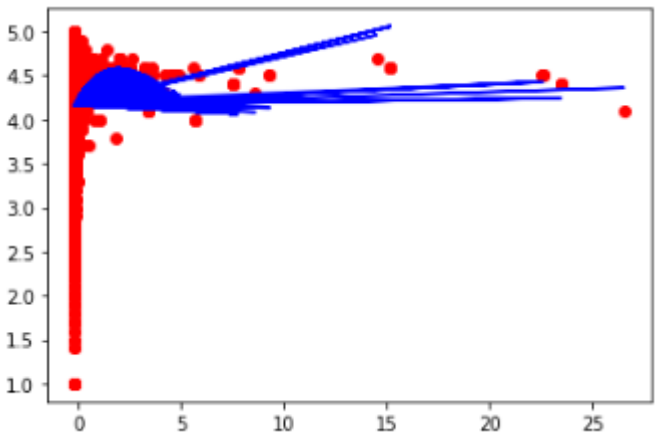


Fig: Plot of Polynomial Regression on Reviews for Degree 5

```
import statsmodels.api as sm
X_train_sm=sm.add_constant(X_polynom)
lr=sm.OLS(y_train,X_train_sm).fit()
lr.params
```

```
const    4.231012
x1       0.411793
x2      -0.150466
x3       0.017657
x4      -0.000798
x5       0.000012
dtype: float64
```

Fig: Best parameter for Degree 5

So, here we have applied polynomial regression for degree 5 and it is clearly visible that for higher degree it nearly matches all the actual values. We have also shown the parameters on which this situation is achievable. With the help of OLS, we have also done the deep analysis on this and have shown the complete summary in next figure.

OLS Regression Results

Dep. Variable:	Rating	**R-squared:**	0.025
Model:	OLS	**Adj. R-squared:**	0.025
Method:	Least Squares	**F-statistic:**	39.16
Date:	Sat, 05 Feb 2022	**Prob (F-statistic):**	7.55e-40
Time:	05:43:53	**Log-Likelihood:**	-5084.9
No. Observations:	7587	**AIC:**	1.018e+04
Df Residuals:	7581	**BIC:**	1.022e+04
Df Model:	5		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[0.025	0.975]
const	4.2310	0.007	641.613	0.000	4.218	4.244
x1	0.4118	0.033	12.555	0.000	0.347	0.476
x2	-0.1505	0.017	-8.994	0.000	-0.183	-0.118
x3	0.0177	0.002	7.470	0.000	0.013	0.022
x4	-0.0008	0.000	-6.491	0.000	-0.001	-0.001
x5	1.223e-05	2.13e-06	5.747	0.000	8.06e-06	1.64e-05

Omnibus:	3227.896	**Durbin-Watson:**	2.010
Prob(Omnibus):	0.000	**Jarque-Bera (JB):**	21024.200
Skew:	-1.921	**Prob(JB):**	0.00
Kurtosis:	10.193	**Cond. No.**	1.60e+06

Fig: OLS summary for Reviews after applying Polynomial Regression

- For Installs

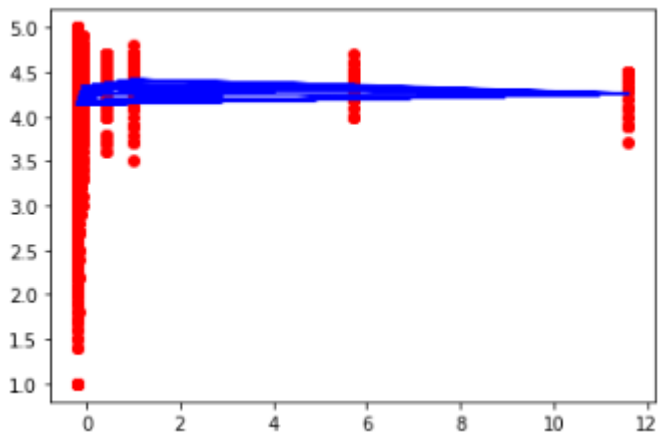


Fig: Plot of polynomial regression on Installs for Degree 5

```
import statsmodels.api as sm
X_train_sm=sm.add_constant(X_polynom)
lr=sm.OLS(y_train,X_train_sm).fit()
lr.params
```

```
const    4.387873
x1       0.705612
x2      -3.079951
x3       2.994885
x4      -0.626844
x5       0.033743
dtype: float64
```

Fig: Best parameter for Degree 5

So, here we have applied polynomial regression for degree 5 and it is clearly visible that for higher degree it nearly matches all the actual values. We have also shown the parameters on which this situation is achievable. With the help of OLS, we have also done the deep analysis on this and have shown the complete summary in next figure.

OLS Regression Results

Dep. Variable:	Rating	**R-squared:**	0.029
Model:	OLS	**Adj. R-squared:**	0.029
Method:	Least Squares	**F-statistic:**	45.94
Date:	Sat, 05 Feb 2022	**Prob (F-statistic):**	6.49e-47
Time:	05:44:05	**Log-Likelihood:**	-5068.4
No. Observations:	7587	**AIC:**	1.015e+04
Df Residuals:	7581	**BIC:**	1.019e+04
Df Model:	5		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[0.025	0.975]
const	4.3879	0.018	238.212	0.000	4.352	4.424
x1	0.7056	0.067	10.543	0.000	0.574	0.837
x2	-3.0800	0.369	-8.354	0.000	-3.803	-2.357
x3	2.9949	0.441	6.788	0.000	2.130	3.860
x4	-0.6268	0.097	-6.492	0.000	-0.816	-0.438
x5	0.0337	0.005	6.392	0.000	0.023	0.044

Omnibus:	3098.363	**Durbin-Watson:**	2.014
Prob(Omnibus):	0.000	**Jarque-Bera (JB):**	19467.485
Skew:	-1.842	**Prob(JB):**	0.00
Kurtosis:	9.929	**Cond. No.**	1.56e+06

Fig: OLS summary for Installs after applying polynomial Regression

- For Size

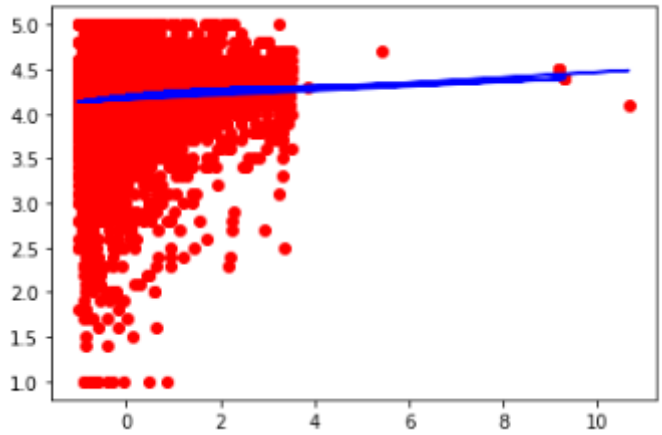


Fig: Plot of polynomial regression on Size for Degree 3

```
import statsmodels.api as sm
X_train_sm=sm.add_constant(X_polynom)
lr=sm.OLS(y_train,X_train_sm).fit()
lr.params
```

```
const    4.201307
x1       0.058501
x2      -0.011666
x3       0.000814
dtype: float64
```

Fig: Best parameter for Degree 3

So, here we have applied polynomial regression for degree 3 and it is clearly visible that for that this time it does not cover all the actual values but this is the best case for degree 3. We have also shown the parameters on which this situation is achievable. With the help of OLS, we have also done the deep analysis on this and have shown the complete summary in next figure.

OLS Regression Results

Dep. Variable:	Rating	**R-squared:**	0.009
Model:	OLS	**Adj. R-squared:**	0.008
Method:	Least Squares	**F-statistic:**	21.85
Date:	Sat, 05 Feb 2022	**Prob (F-statistic):**	4.37e-14
Time:	05:44:15	**Log-Likelihood:**	-5149.0
No. Observations:	7587	**AIC:**	1.031e+04
Df Residuals:	7583	**BIC:**	1.033e+04
Df Model:	3		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[0.025	0.975]
const	4.2013	0.007	585.974	0.000	4.187	4.215
x1	0.0585	0.009	6.824	0.000	0.042	0.075
x2	-0.0117	0.006	-1.967	0.049	-0.023	-4.14e-05
x3	0.0008	0.001	1.315	0.189	-0.000	0.002

Omnibus:	3234.952	**Durbin-Watson:**	2.011
Prob(Omnibus):	0.000	**Jarque-Bera (JB):**	20551.533
Skew:	-1.936	**Prob(JB):**	0.00
Kurtosis:	10.072	**Cond. No.**	53.7

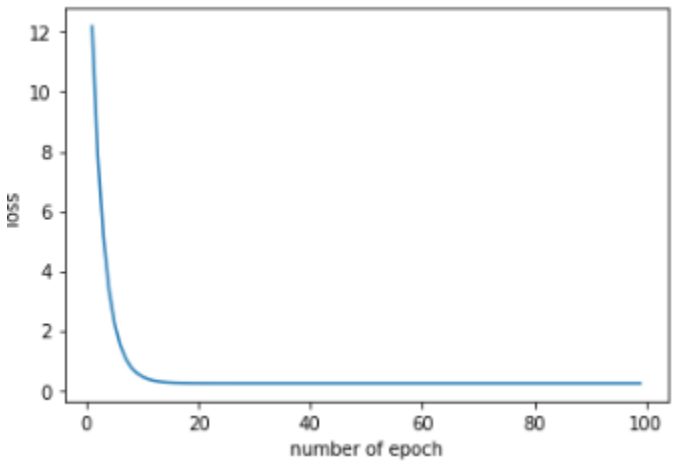
Fig: OLS summary for Size after applying polynomial Regression

4.4. Gradient Descent Method

(Learning Rate: 0.1, 0.01, 0.001, 0.5, 0.05, 0.005, 1, Iterations 100, 500, 1000)

For Reviews

- (100,0.1)




```
print("weight:",w)
print("bias:",b)
```

```
weight: [0.03542703]
bias: 4.191096101380706
```

```
def predict(inp):
    y_lst=[]
    for i in range(len(inp)):
        y_lst.append(w@inp[i]+b)
    return np.array(y_lst)
```

```
X_test=sc.fit_transform(x_test)
y_pred=predict(X_test)
```

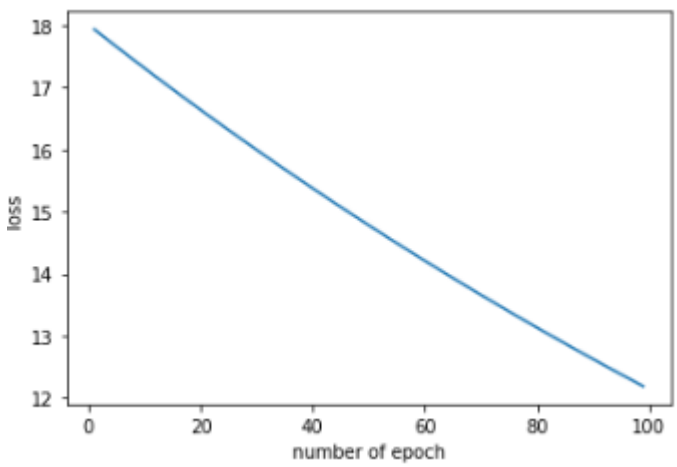
```
from sklearn.metrics import mean_squared_error, r2_score, mean_absolute_error
from math import sqrt
print("R2 score : %.3f" % r2_score(y_test, y_pred))
print("MSE: %.3f" % mean_squared_error(y_test, y_pred))
print("RMSE: %.3f" % sqrt(mean_squared_error(y_test, y_pred)))
print("MAE: %.3f" % sqrt(mean_absolute_error(y_test, y_pred)))
```

```
R2 score : 0.003
MSE: 0.229
RMSE: 0.479
MAE: 0.560
```

Fig: Gradient Plot for (100,0.1) and calculation of errors

Here the gradient function (Loss or Cost) gradually decreases and after a point become constant and all kind of error values are calculated.

- (100,0.001)



```
print("weight:",w)
print("bias:",b)
```

```
weight: [0.41474839]
bias: 0.7604039576862713
```

```
def predict(inp):
    y_lst=[]
    for i in range(len(inp)):
        y_lst.append(w@inp[i]+b)
    return np.array(y_lst)
```

```
X_test=sc.fit_transform(x_test)
y_pred=predict(X_test)
```

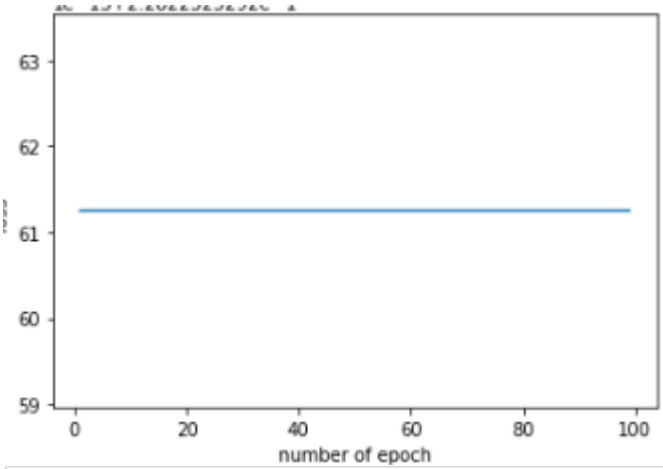
```
from sklearn.metrics import mean_squared_error, r2_score, mean_absolute_error
from math import sqrt
print("R2 score : %.3f" % r2_score(y_test, y_pred))
print("MSE: %.3f" % mean_squared_error(y_test, y_pred))
print("RMSE: %.3f" % sqrt(mean_squared_error(y_test, y_pred)))
print("MAE: %.3f" % sqrt(mean_absolute_error(y_test, y_pred)))
```

```
R2 score : -51.719
MSE: 12.120
RMSE: 3.481
MAE: 1.855
```

Fig: Gradient Plot for (100,0.001) and calculation of errors

Here the gradient function (Loss or Cost) decreases in a straight line in a fast manner and does not become constant and all kind of error values are calculated.

- (100,0.5)




```
print("weight:",w)
print("bias:",b)
```

```
weight: [0.03542703]
bias: 4.1910961022344475
```

```
def predict(inp):
    y_lst=[]
    for i in range(len(inp)):
        y_lst.append(w@inp[i]+b)
    return np.array(y_lst)
```

```
X_test=sc.fit_transform(x_test)
y_pred=predict(X_test)
```

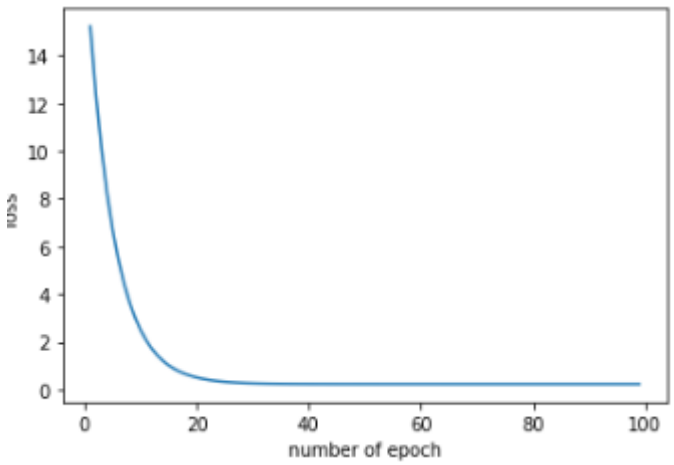
```
from sklearn.metrics import mean_squared_error, r2_score, mean_absolute_error
from math import sqrt
print("R2 score : %.3f" % r2_score(y_test, y_pred))
print("MSE: %.3f" % mean_squared_error(y_test, y_pred))
print("RMSE: %.3f" % sqrt(mean_squared_error(y_test, y_pred)))
print("MAE: %.3f" % sqrt(mean_absolute_error(y_test, y_pred)))
```

```
R2 score : 0.003
MSE: 0.229
RMSE: 0.479
MAE: 0.560
```

Fig: Gradient Plot for (100,0.5) and calculation of errors

Here the gradient function (Loss or Cost) is nearly constant for all points and all kind of error values are calculated.

- (100,0.05)




```
print("weight:",w)
print("bias:",b)
```

```
weight: [0.03540115]
bias: 4.190984780859099
```

```
def predict(inp):
    y_lst=[]
    for i in range(len(inp)):
        y_lst.append(w@inp[i]+b)
    return np.array(y_lst)
```

```
X_test=sc.fit_transform(x_test)
y_pred=predict(X_test)
```

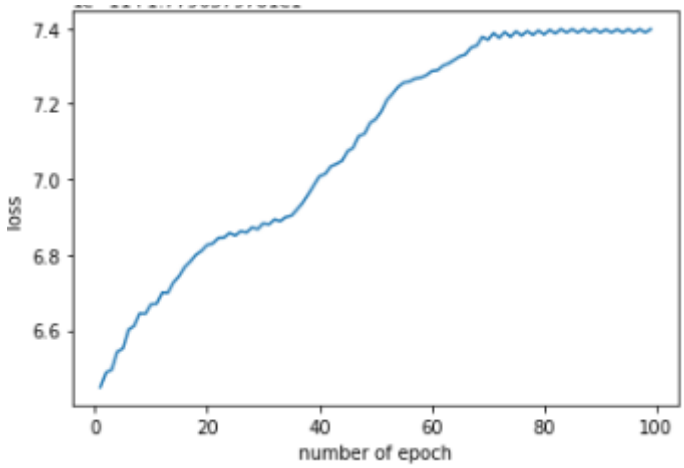
```
from sklearn.metrics import mean_squared_error, r2_score, mean_absolute_error
from math import sqrt
print("R2 score : %.3f" % r2_score(y_test, y_pred))
print("MSE: %.3f" % mean_squared_error(y_test, y_pred))
print("RMSE: %.3f" % sqrt(mean_squared_error(y_test, y_pred)))
print("MAE: %.3f" % sqrt(mean_absolute_error(y_test, y_pred)))
```

```
R2 score : 0.003
MSE: 0.229
RMSE: 0.479
MAE: 0.560
```

Fig: Gradient Plot for (100,0.05) and calculation of errors

Here the gradient function (Loss or Cost) gradually decreases and after a point become constant and all kind of error values are calculated.

- (100, 1)



```
print("weight:",w)
print("bias:",b)
```

```
weight: [0.0889995]
bias: -1.127986593019159e-12
```

```
def predict(inp):
    y_lst=[]
    for i in range(len(inp)):
        y_lst.append(w@inp[i]+b)
    return np.array(y_lst)
```

```
X_test=sc.fit_transform(x_test)
y_pred=predict(X_test)
```

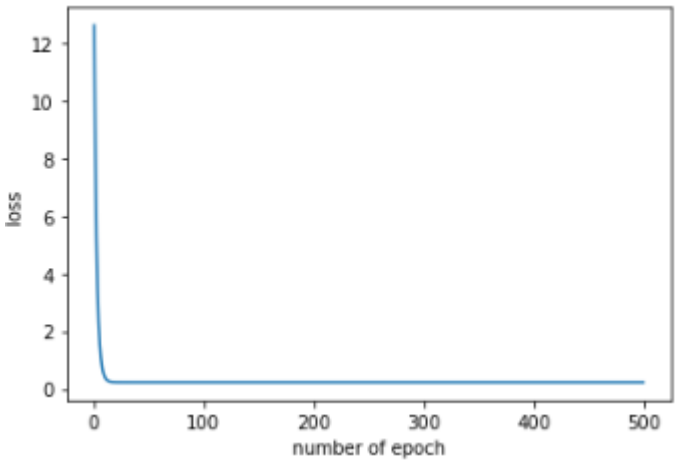
```
from sklearn.metrics import mean_squared_error, r2_score, mean_absolute_error
from math import sqrt
print("R2 score : %.3f" % r2_score(y_test, y_pred))
print("MSE: %.3f" % mean_squared_error(y_test, y_pred))
print("RMSE: %.3f" % sqrt(mean_squared_error(y_test, y_pred)))
print("MAE: %.3f" % sqrt(mean_absolute_error(y_test, y_pred)))
```

```
R2 score : -76.267
MSE: 17.764
RMSE: 4.215
MAE: 2.046
```

Fig: Gradient Plot for (100,1) and calculation of errors

Here the gradient function (Loss or Cost) gradually increases due to high learning rate and after a point become constant and all kind of error values are calculated.

- (500,0.1)




```
print("weight:",w)
print("bias:",b)
```

```
weight: [0.03542703]
bias: 4.191096102234446
```

```
def predict(inp):
    y_lst=[]
    for i in range(len(inp)):
        y_lst.append(w@inp[i]+b)
    return np.array(y_lst)
```

```
X_test=sc.fit_transform(x_test)
y_pred=predict(X_test)
```

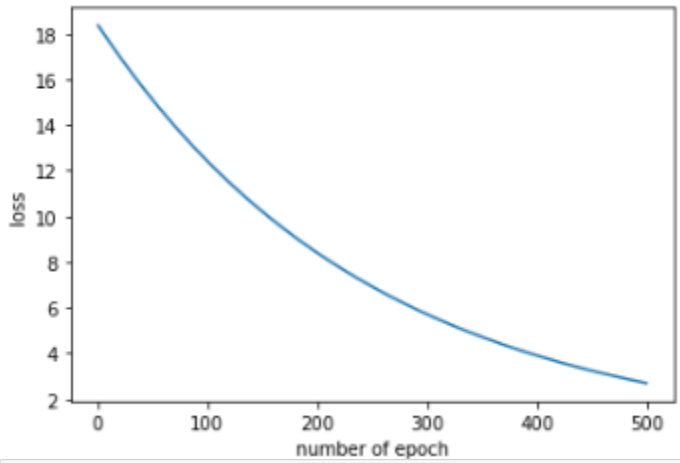
```
from sklearn.metrics import mean_squared_error, r2_score, mean_absolute_error
from math import sqrt
print("R2 score : %.3f" % r2_score(y_test, y_pred))
print("MSE: %.3f" % mean_squared_error(y_test, y_pred))
print("RMSE: %.3f" % sqrt(mean_squared_error(y_test, y_pred)))
print("MAE: %.3f" % sqrt(mean_absolute_error(y_test, y_pred)))
```

```
R2 score : 0.003
MSE: 0.229
RMSE: 0.479
MAE: 0.560
```

Fig: Gradient Plot for (500,0.1) and calculation of errors

Here the gradient function (Loss or Cost) gradually decreases and after a point become constant and all kind of error values are calculated.

- (500,0.001)




```
print("weight:",w)
print("bias:",b)
```

```
weight: [-0.25707248]
bias: 2.6508211144753164
```

```
def predict(inp):
    y_lst=[]
    for i in range(len(inp)):
        y_lst.append(w@inp[i]+b)
    return np.array(y_lst)
```

```
X_test=sc.fit_transform(x_test)
y_pred=predict(X_test)
```

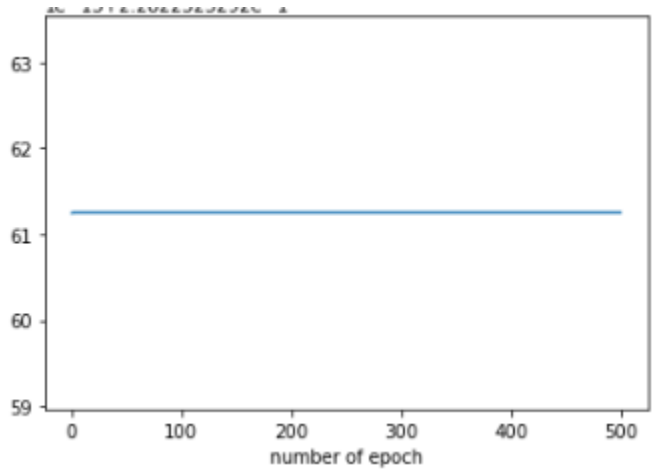
```
from sklearn.metrics import mean_squared_error, r2_score, mean_absolute_error
from math import sqrt
print("R2 score : %.3f" % r2_score(y_test, y_pred))
print("MSE: %.3f" % mean_squared_error(y_test, y_pred))
print("RMSE: %.3f" % sqrt(mean_squared_error(y_test, y_pred)))
print("MAE: %.3f" % sqrt(mean_absolute_error(y_test, y_pred)))
```

```
R2 score : -10.614
MSE: 2.670
RMSE: 1.634
MAE: 1.247
```

Fig: Gradient Plot for (500,0.001) and calculation of errors

Here the gradient function (Loss or Cost) decreases in a curve path and all kind of error values are calculated.

- (500,0.5)



```
print("weight:",w)
print("bias:",b)
```

```
weight: [0.03542703]
bias: 4.1910961022344475
```

```
def predict(inp):
    y_lst=[]
    for i in range(len(inp)):
        y_lst.append(w@inp[i]+b)
    return np.array(y_lst)
```

```
X_test=sc.fit_transform(x_test)
y_pred=predict(X_test)
```

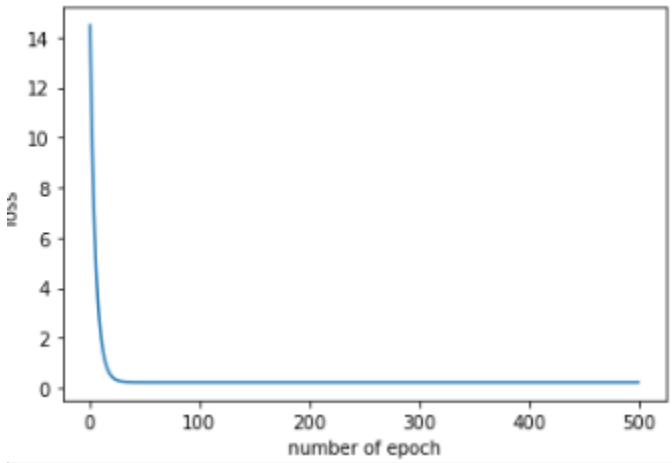
```
from sklearn.metrics import mean_squared_error, r2_score, mean_absolute_error
from math import sqrt
print("R2 score : %.3f" % r2_score(y_test, y_pred))
print("MSE: %.3f" % mean_squared_error(y_test, y_pred))
print("RMSE: %.3f" % sqrt(mean_squared_error(y_test, y_pred)))
print("MAE: %.3f" % sqrt(mean_absolute_error(y_test, y_pred)))
```

```
R2 score : 0.003
MSE: 0.229
RMSE: 0.479
MAE: 0.560
```

Fig: Gradient Plot for (500,0.5) and calculation of errors

Here the gradient function (Loss or Cost) is almost constant for each point and all kind of error values are calculated.

- (500,0.05)




```
print("weight:",w)
print("bias:",b)
```

```
weight: [0.03542703]
bias: 4.191096102234443
```

```
def predict(inp):
    y_lst=[]
    for i in range(len(inp)):
        y_lst.append(w@inp[i]+b)
    return np.array(y_lst)
```

```
X_test=sc.fit_transform(x_test)
y_pred=predict(X_test)
```

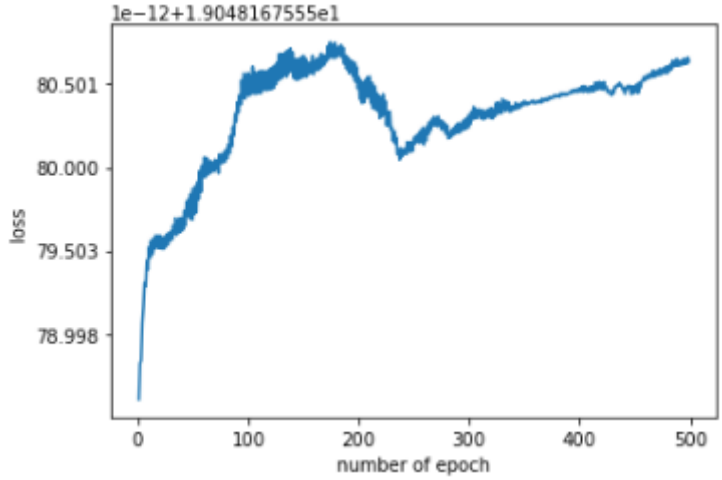
```
from sklearn.metrics import mean_squared_error, r2_score, mean_absolute_error
from math import sqrt
print("R2 score : %.3f" % r2_score(y_test, y_pred))
print("MSE: %.3f" % mean_squared_error(y_test, y_pred))
print("RMSE: %.3f" % sqrt(mean_squared_error(y_test, y_pred)))
print("MAE: %.3f" % sqrt(mean_absolute_error(y_test, y_pred)))
```

```
R2 score : 0.003
MSE: 0.229
RMSE: 0.479
MAE: 0.560
```

Fig: Gradient Plot for (500,0.05) and calculation of errors

Here the gradient function (Loss or Cost) gradually decreases and after a point become constant and all kind of error values are calculated.

- (500,1)




```
print("weight:",w)
print("bias:",b)
```

```
weight: [1.1555421]
bias: 9.592326932761353e-14
```

```
def predict(inp):
    y_lst=[]
    for i in range(len(inp)):
        y_lst.append(w@inp[i]+b)
    return np.array(y_lst)
```

```
X_test=sc.fit_transform(x_test)
y_pred=predict(X_test)
```

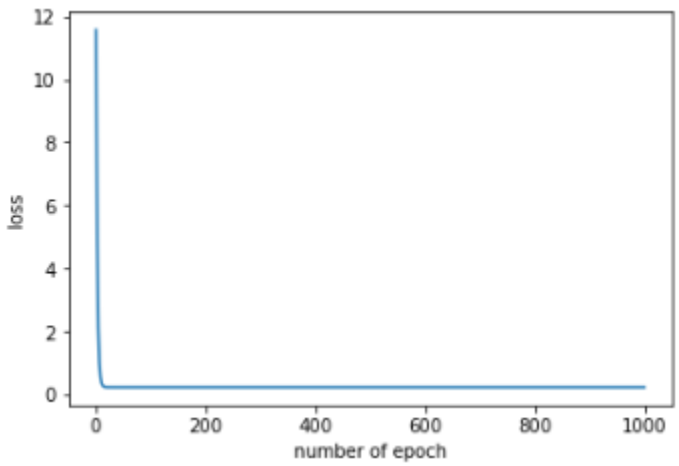
```
from sklearn.metrics import mean_squared_error, r2_score, mean_absolute_error
from math import sqrt
print("R2 score : %.3f" % r2_score(y_test, y_pred))
print("MSE: %.3f" % mean_squared_error(y_test, y_pred))
print("RMSE: %.3f" % sqrt(mean_squared_error(y_test, y_pred)))
print("MAE: %.3f" % sqrt(mean_absolute_error(y_test, y_pred)))
```

```
R2 score : -81.781
MSE: 19.032
RMSE: 4.363
MAE: 2.069
```

Fig: Gradient Plot for (500,1) and calculation of errors

Here the gradient function (Loss or Cost) gradually increases and then decreases for some time and then again increases and all kind of error values are calculated.

- (1000,0.1)



```
print("weight:",w)
print("bias:",b)
```

```
weight: [0.03542703]
bias: 4.191096102234446
```

```
def predict(inp):
    y_lst=[]
    for i in range(len(inp)):
        y_lst.append(w@inp[i]+b)
    return np.array(y_lst)
```

```
X_test=sc.fit_transform(x_test)
y_pred=predict(X_test)
```

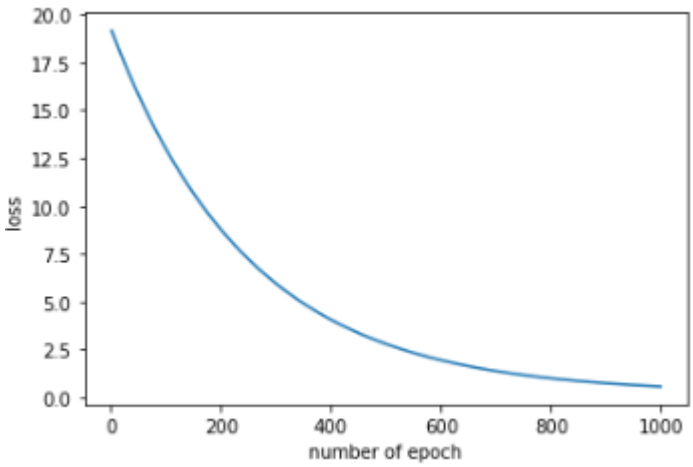
```
from sklearn.metrics import mean_squared_error, r2_score, mean_absolute_error
from math import sqrt
print("R2 score : %.3f" % r2_score(y_test, y_pred))
print("MSE: %.3f" % mean_squared_error(y_test, y_pred))
print("RMSE: %.3f" % sqrt(mean_squared_error(y_test, y_pred)))
print("MAE: %.3f" % sqrt(mean_absolute_error(y_test, y_pred)))
```

```
R2 score : 0.003
MSE: 0.229
RMSE: 0.479
MAE: 0.560
```

Fig: Gradient Plot for (1000,0.1) and calculation of errors

Here the gradient function (Loss or Cost) gradually decreases and after a point become constant and all kind of error values are calculated.

- (1000,0.001)




```
print("weight:",w)
print("bias:",b)
```

```
weight: [0.1976773]
bias: 3.6250277086579956
```

```
def predict(inp):
    y_lst=[]
    for i in range(len(inp)):
        y_lst.append(w@inp[i]+b)
    return np.array(y_lst)
```

```
X_test=sc.fit_transform(x_test)
y_pred=predict(X_test)
```

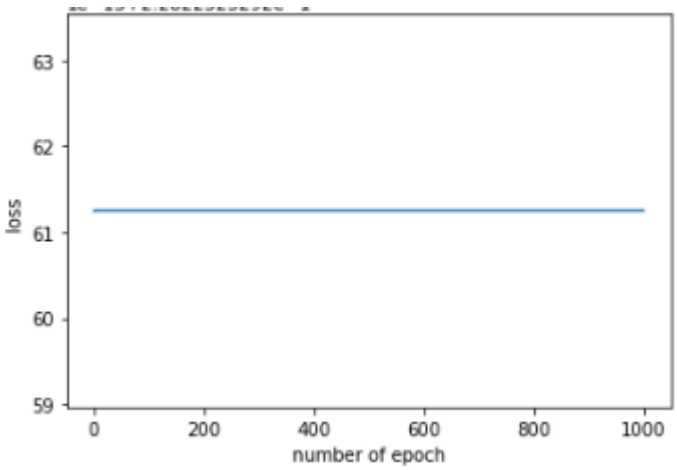
```
from sklearn.metrics import mean_squared_error, r2_score, mean_absolute_error
from math import sqrt
print("R2 score : %.3f" % r2_score(y_test, y_pred))
print("MSE: %.3f" % mean_squared_error(y_test, y_pred))
print("RMSE: %.3f" % sqrt(mean_squared_error(y_test, y_pred)))
print("MAE: %.3f" % sqrt(mean_absolute_error(y_test, y_pred)))
```

```
R2 score : -1.496
MSE: 0.574
RMSE: 0.757
MAE: 0.823
```

Fig: Gradient Plot for (1000,0.001) and calculation of errors

Here the gradient function (Loss or Cost) decreases in a curve path and after a point become constant and all kind of error values are calculated.

- (1000,0.5)




```
print("weight:",w)
print("bias:",b)
```

```
weight: [0.03542703]
bias: 4.1910961022344475
```

```
def predict(inp):
    y_lst=[]
    for i in range(len(inp)):
        y_lst.append(w@inp[i]+b)
    return np.array(y_lst)
```

```
X_test=sc.fit_transform(x_test)
y_pred=predict(X_test)
```

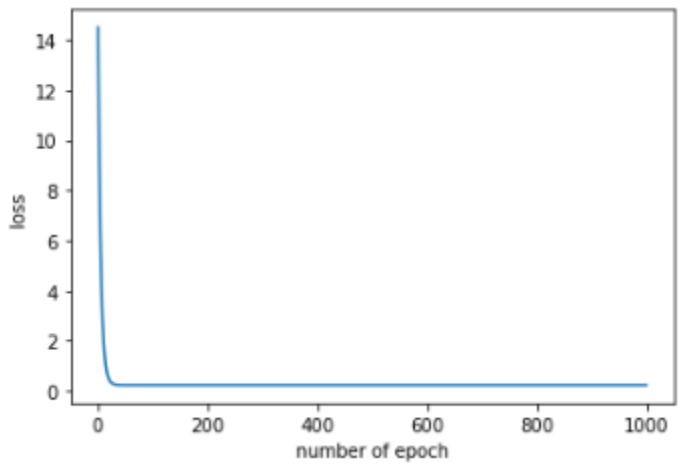
```
from sklearn.metrics import mean_squared_error, r2_score, mean_absolute_error
from math import sqrt
print("R2 score : %.3f" % r2_score(y_test, y_pred))
print("MSE: %.3f" % mean_squared_error(y_test, y_pred))
print("RMSE: %.3f" % sqrt(mean_squared_error(y_test, y_pred)))
print("MAE: %.3f" % sqrt(mean_absolute_error(y_test, y_pred)))
```

```
R2 score : 0.003
MSE: 0.229
RMSE: 0.479
MAE: 0.560
```

Fig: Gradient Plot for (1000,0.5) and calculation of errors

Here the gradient function (Loss or Cost) is almost constant throughout the period and all kind of error values are calculated.

- (1000,0.05)



```
print("weight:",w)
print("bias:",b)
```

```
weight: [0.03542703]
bias: 4.191096102234443
```

```
def predict(inp):
    y_lst=[]
    for i in range(len(inp)):
        y_lst.append(w@inp[i]+b)
    return np.array(y_lst)
```

```
X_test=sc.fit_transform(x_test)
y_pred=predict(X_test)
```

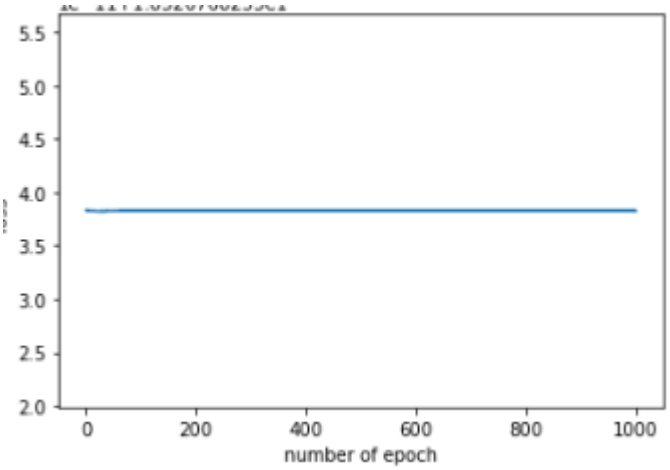
```
from sklearn.metrics import mean_squared_error, r2_score, mean_absolute_error
from math import sqrt
print("R2 score : %.3f" % r2_score(y_test, y_pred))
print("MSE: %.3f" % mean_squared_error(y_test, y_pred))
print("RMSE: %.3f" % sqrt(mean_squared_error(y_test, y_pred)))
print("MAE: %.3f" % sqrt(mean_absolute_error(y_test, y_pred)))
```

```
R2 score : 0.003
MSE: 0.229
RMSE: 0.479
MAE: 0.560
```

Fig: Gradient Plot for (1000,0.05) and calculation of errors

Here the gradient function (Loss or Cost) gradually decreases and after a point become constant and all kind of error values are calculated.

- (1000,1)




```
print("weight:",w)
print("bias:",b)
```

```
weight: [-0.69481845]
bias: 1.5987211554602254e-14
```

```
def predict(inp):
    y_lst=[]
    for i in range(len(inp)):
        y_lst.append(w@inp[i]+b)
    return np.array(y_lst)
```

```
X_test=sc.fit_transform(x_test)
y_pred=predict(X_test)
```

```
from sklearn.metrics import mean_squared_error, r2_score, mean_absolute_error
from math import sqrt
print("R2 score : %.3f" % r2_score(y_test, y_pred))
print("MSE: %.3f" % mean_squared_error(y_test, y_pred))
print("RMSE: %.3f" % sqrt(mean_squared_error(y_test, y_pred)))
print("MAE: %.3f" % sqrt(mean_absolute_error(y_test, y_pred)))
```

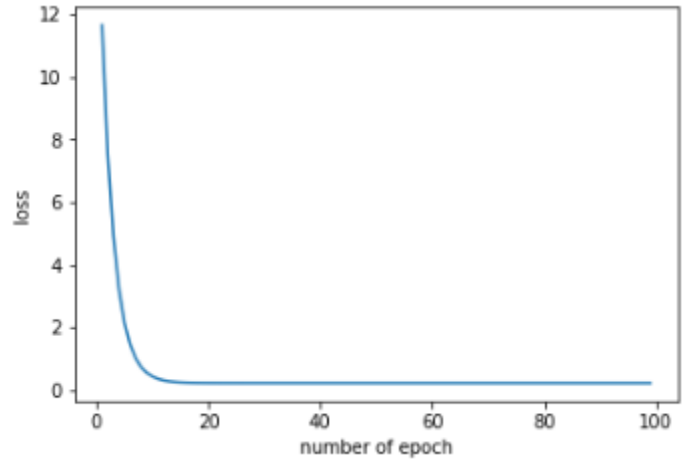
```
R2 score : -78.522
MSE: 18.283
RMSE: 4.276
MAE: 2.046
```

Fig: Gradient Plot for (1000,1) and calculation of errors

Here the gradient function (Loss or Cost) is constant for the entire time and all kind of error values are calculated.

For Installs

- (100,0.1)




```
print("weight:",w)
print("bias:",b)
```

```
weight: [0.02532093]
bias: 4.1910961013807055
```

```
def predict(inp):
    y_lst=[]
    for i in range(len(inp)):
        y_lst.append(w@inp[i]+b)
    return np.array(y_lst)
```

```
X_test=sc.fit_transform(x_test)
y_pred=predict(X_test)
```

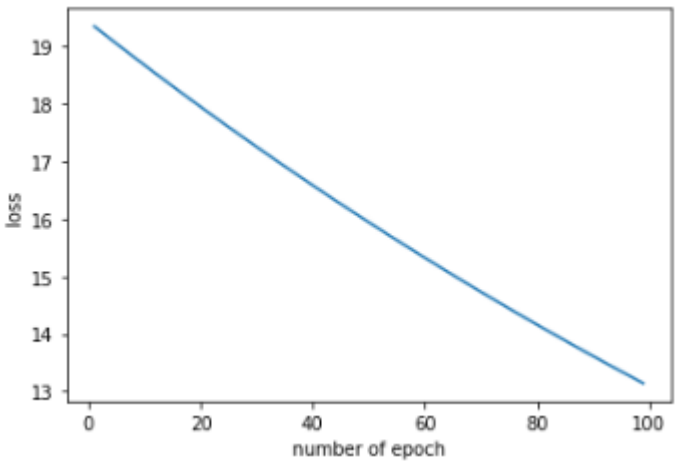
```
from sklearn.metrics import mean_squared_error, r2_score, mean_absolute_error
from math import sqrt
print("R2 score : %.3f" % r2_score(y_test, y_pred))
print("MSE: %.3f" % mean_squared_error(y_test, y_pred))
print("RMSE: %.3f" % sqrt(mean_squared_error(y_test, y_pred)))
print("MAE: %.3f" % sqrt(mean_absolute_error(y_test, y_pred)))
```

```
R2 score : 0.002
MSE: 0.229
RMSE: 0.479
MAE: 0.561
```

Fig: Gradient Plot for (100,0.1) and calculation of errors

Here the gradient function (Loss or Cost) gradually decreases and after a point become constant and all kind of error values are calculated.

- (100,0.001)



```
print("weight:",w)
print("bias:",b)
```

```
weight: [-1.0204976]
bias: 0.7604039576862708
```

```
def predict(inp):
    y_lst=[]
    for i in range(len(inp)):
        y_lst.append(w@inp[i]+b)
    return np.array(y_lst)
```

```
X_test=sc.fit_transform(x_test)
y_pred=predict(X_test)
```

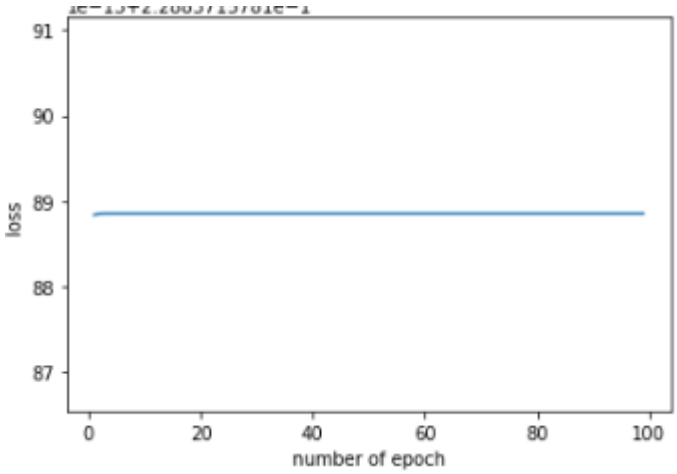
```
from sklearn.metrics import mean_squared_error, r2_score, mean_absolute_error
from math import sqrt
print("R2 score : %.3f" % r2_score(y_test, y_pred))
print("MSE: %.3f" % mean_squared_error(y_test, y_pred))
print("RMSE: %.3f" % sqrt(mean_squared_error(y_test, y_pred)))
print("MAE: %.3f" % sqrt(mean_absolute_error(y_test, y_pred)))
```

```
R2 score : -55.815
MSE: 13.062
RMSE: 3.614
MAE: 1.851
```

Fig: Gradient Plot for (100,0.001) and calculation of errors

Here the gradient function (Loss or Cost) decreases in a straight line and does not become constant and all kind of error values are calculated.

- (100,0.5)




```
print("weight:",w)
print("bias:",b)
```

```
weight: [0.02532093]
bias: 4.1910961022344475
```

```
def predict(inp):
    y_lst=[]
    for i in range(len(inp)):
        y_lst.append(w@inp[i]+b)
    return np.array(y_lst)
```

```
X_test=sc.fit_transform(x_test)
y_pred=predict(X_test)
```

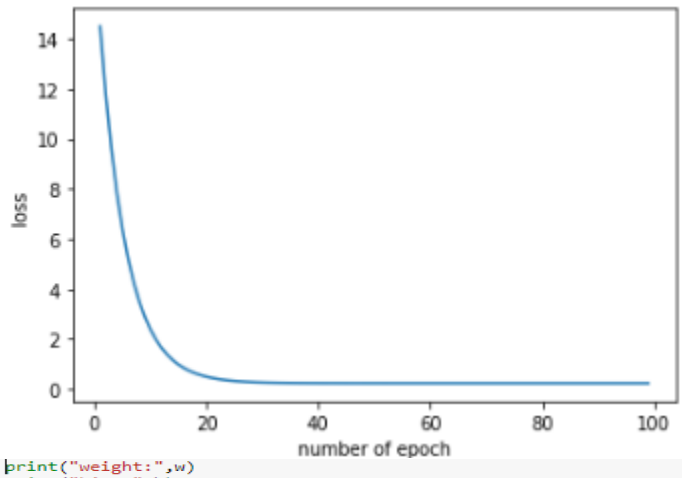
```
from sklearn.metrics import mean_squared_error, r2_score, mean_absolute_error
from math import sqrt
print("R2 score : %.3f" % r2_score(y_test, y_pred))
print("MSE: %.3f" % mean_squared_error(y_test, y_pred))
print("RMSE: %.3f" % sqrt(mean_squared_error(y_test, y_pred)))
print("MAE: %.3f" % sqrt(mean_absolute_error(y_test, y_pred)))
```

```
R2 score : 0.002
MSE: 0.229
RMSE: 0.479
MAE: 0.561
```

Fig: Gradient Plot for (100,0.5) and calculation of errors

Here the gradient function (Loss or Cost) is constant for whole period of time and all kind of error values are calculated.

- (100,0.05)




```
print("weight:",w)
print("bias:",b)
```

```
weight: [0.02531469]
bias: 4.190984780859099
```

```
def predict(inp):
    y_lst=[]
    for i in range(len(inp)):
        y_lst.append(w@inp[i]+b)
    return np.array(y_lst)
```

```
X_test=sc.fit_transform(x_test)
y_pred=predict(X_test)
```

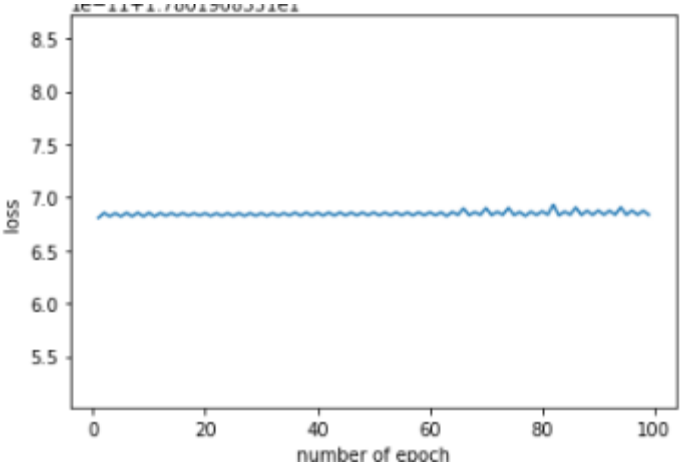
```
from sklearn.metrics import mean_squared_error, r2_score, mean_absolute_error
from math import sqrt
print("R2 score : %.3f" % r2_score(y_test, y_pred))
print("MSE: %.3f" % mean_squared_error(y_test, y_pred))
print("RMSE: %.3f" % sqrt(mean_squared_error(y_test, y_pred)))
print("MAE: %.3f" % sqrt(mean_absolute_error(y_test, y_pred)))
```

```
R2 score : 0.002
MSE: 0.229
RMSE: 0.479
MAE: 0.561
```

Fig: Gradient Plot for (100,0.05) and calculation of errors

Here the gradient function (Loss or Cost) gradually decreases and after a point become constant and all kind of error values are calculated.

- (100, 1)



```
print("weight:",w)
print("bias:",b)
```

```
weight: [-0.06290965]
bias: -7.815970093361102e-14
```

```
def predict(inp):
    y_lst=[]
    for i in range(len(inp)):
        y_lst.append(w@inp[i]+b)
    return np.array(y_lst)
```

```
X_test=sc.fit_transform(x_test)
y_pred=predict(X_test)
```

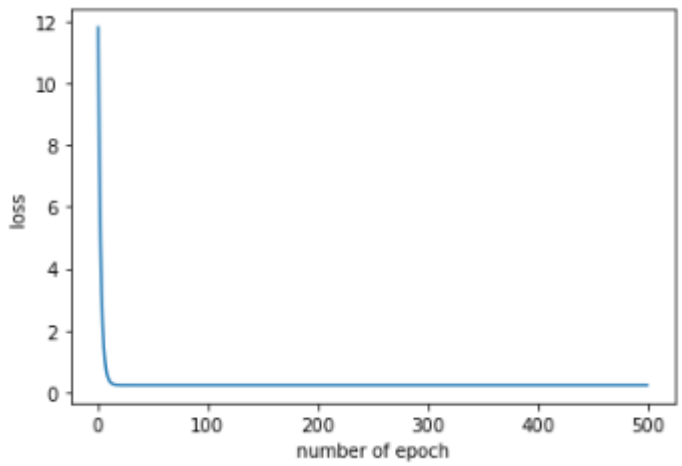
```
from sklearn.metrics import mean_squared_error, r2_score, mean_absolute_error
from math import sqrt
print("R2 score : %.3f" % r2_score(y_test, y_pred))
print("MSE: %.3f" % mean_squared_error(y_test, y_pred))
print("RMSE: %.3f" % sqrt(mean_squared_error(y_test, y_pred)))
print("MAE: %.3f" % sqrt(mean_absolute_error(y_test, y_pred)))
```

```
R2 score : -76.284
MSE: 17.768
RMSE: 4.215
MAE: 2.046
```

Fig: Gradient Plot for (100,1) and calculation of errors

Here the gradient function (Loss or Cost) is almost constant just at the end a little bit of fluctuation is present and all kind of error values are calculated.

- (500,0.1)




```
print("weight:",w)
print("bias:",b)
```

```
weight: [0.02532093]
bias: 4.191096102234446
```

```
def predict(inp):
    y_lst=[]
    for i in range(len(inp)):
        y_lst.append(w@inp[i]+b)
    return np.array(y_lst)
```

```
X_test=sc.fit_transform(x_test)
y_pred=predict(X_test)
```

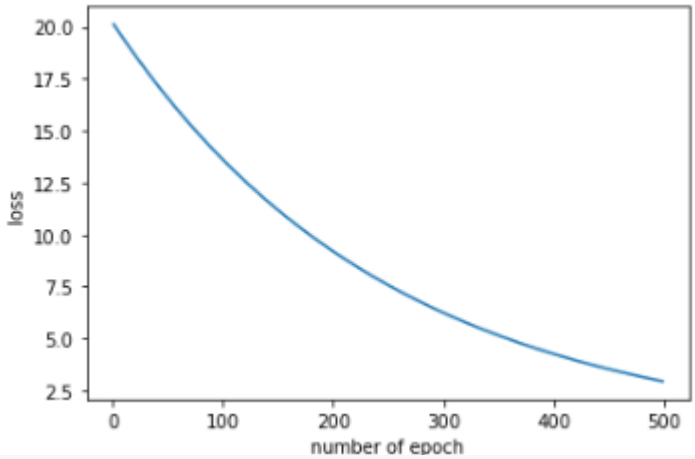
```
from sklearn.metrics import mean_squared_error, r2_score, mean_absolute_error
from math import sqrt
print("R2 score : %.3f" % r2_score(y_test, y_pred))
print("MSE: %.3f" % mean_squared_error(y_test, y_pred))
print("RMSE: %.3f" % sqrt(mean_squared_error(y_test, y_pred)))
print("MAE: %.3f" % sqrt(mean_absolute_error(y_test, y_pred)))
```

```
R2 score : 0.002
MSE: 0.229
RMSE: 0.479
MAE: 0.561
```

Fig: Gradient Plot for (500,0.1) and calculation of errors

Here the gradient function (Loss or Cost) gradually decreases and after a point become constant and all kind of error values are calculated.

- (500,0.001)




```
print("weight:",w)
print("bias:",b)
```

```
weight: [0.59540455]
bias: 2.6508211144753173
```

```
def predict(inp):
    y_lst=[]
    for i in range(len(inp)):
        y_lst.append(w@inp[i]+b)
    return np.array(y_lst)
```

```
X_test=sc.fit_transform(x_test)
y_pred=predict(X_test)
```

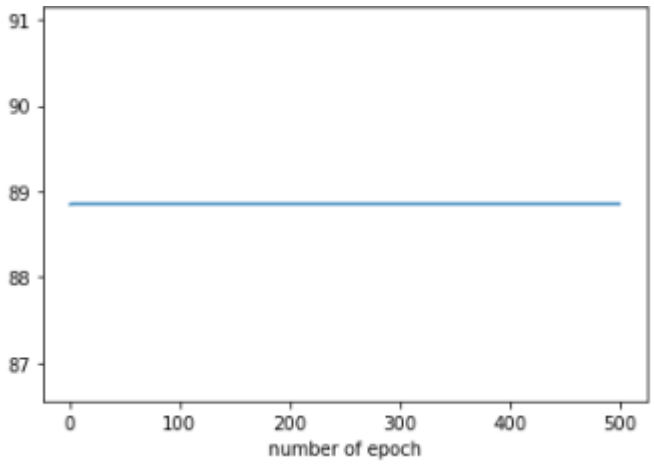
```
from sklearn.metrics import mean_squared_error, r2_score, mean_absolute_error
from math import sqrt
print("R2 score : %.3f" % r2_score(y_test, y_pred))
print("MSE: %.3f" % mean_squared_error(y_test, y_pred))
print("RMSE: %.3f" % sqrt(mean_squared_error(y_test, y_pred)))
print("MAE: %.3f" % sqrt(mean_absolute_error(y_test, y_pred)))
```

```
R2 score : -11.682
MSE: 2.916
RMSE: 1.708
```

Fig: Gradient Plot for (500,0.001) and calculation of errors

Here the gradient function (Loss or Cost) decreases in a curve path and after a point become constant and all kind of error values are calculated.

- (500,0.5)



```
print("weight:",w)
print("bias:",b)
```

```
weight: [0.02532093]
bias: 4.1910961022344475
```

```
def predict(inp):
    y_lst=[]
    for i in range(len(inp)):
        y_lst.append(w@inp[i]+b)
    return np.array(y_lst)
```

```
X_test=sc.fit_transform(x_test)
y_pred=predict(X_test)
```

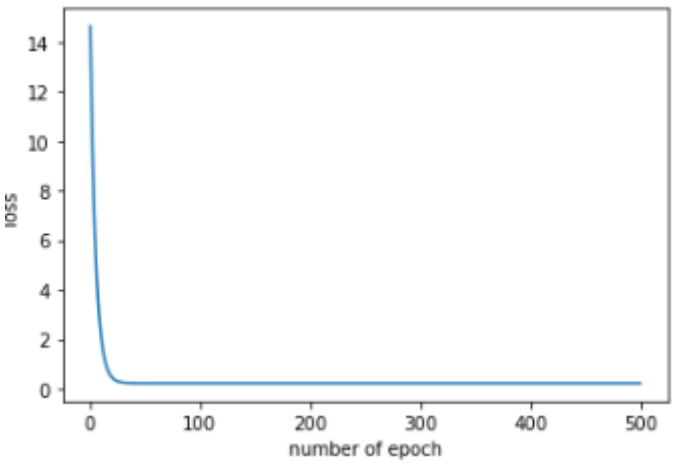
```
from sklearn.metrics import mean_squared_error, r2_score, mean_absolute_error
from math import sqrt
print("R2 score : %.3f" % r2_score(y_test, y_pred))
print("MSE: %.3f" % mean_squared_error(y_test, y_pred))
print("RMSE: %.3f" % sqrt(mean_squared_error(y_test, y_pred)))
print("MAE: %.3f" % sqrt(mean_absolute_error(y_test, y_pred)))
```

```
R2 score : 0.002
MSE: 0.229
RMSE: 0.479
MAE: 0.561
```

Fig: Gradient Plot for (500,0.5) and calculation of errors

Here the gradient function (Loss or Cost) is constant for the whole duration and all kind of error values are calculated.

- (500,0.05)




```
print("weight:",w)
print("bias:",b)
```

```
weight: [0.02532093]
bias: 4.191096102234443
```

```
def predict(inp):
    y_lst=[]
    for i in range(len(inp)):
        y_lst.append(w@inp[i]+b)
    return np.array(y_lst)
```

```
X_test=sc.fit_transform(x_test)
y_pred=predict(X_test)
```

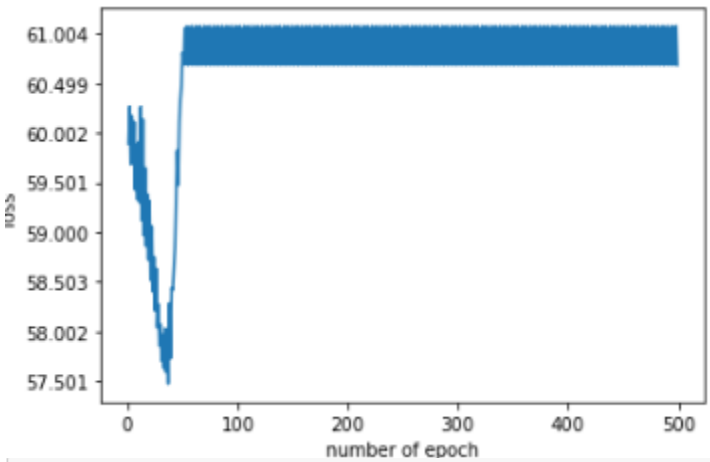
```
from sklearn.metrics import mean_squared_error, r2_score, mean_absolute_error
from math import sqrt
print("R2 score : %.3f" % r2_score(y_test, y_pred))
print("MSE: %.3f" % mean_squared_error(y_test, y_pred))
print("RMSE: %.3f" % sqrt(mean_squared_error(y_test, y_pred)))
print("MAE: %.3f" % sqrt(mean_absolute_error(y_test, y_pred)))
```

```
R2 score : 0.002
MSE: 0.229
RMSE: 0.479
MAE: 0.561
```

Fig: Gradient Plot for (500,0.05) and calculation of errors

Here the gradient function (Loss or Cost) gradually decreases and after a point become constant and all kind of error values are calculated.

- (500,1)




```
print("weight:",w)
print("bias:",b)
```

```
weight: [-1.16054853]
bias: -2.9309887850104133e-13
```

```
def predict(inp):
    y_lst=[]
    for i in range(len(inp)):
        y_lst.append(w@inp[i]+b)
    return np.array(y_lst)
```

```
X_test=sc.fit_transform(x_test)
y_pred=predict(X_test)
```

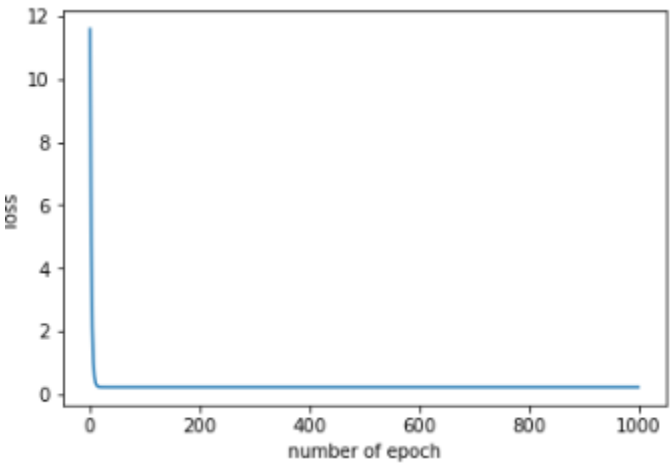
```
from sklearn.metrics import mean_squared_error, r2_score, mean_absolute_error
from math import sqrt
print("R2 score : %.3f" % r2_score(y_test, y_pred))
print("MSE: %.3f" % mean_squared_error(y_test, y_pred))
print("RMSE: %.3f" % sqrt(mean_squared_error(y_test, y_pred)))
print("MAE: %.3f" % sqrt(mean_absolute_error(y_test, y_pred)))
```

```
R2 score : -82.355
MSE: 19.164
RMSE: 4.378
MAE: 2.046
```

Fig: Gradient Plot for (500,1) and calculation of errors

Here the gradient function (Loss or Cost) decreases inclined and then increases for a particular point and after that point it become constant and all kind of error values are calculated.

- (1000,0.1)



```
print("weight:",w)
print("bias:",b)
```

```
weight: [0.02532093]
bias: 4.191096102234446
```

```
def predict(inp):
    y_lst=[]
    for i in range(len(inp)):
        y_lst.append(w@inp[i]+b)
    return np.array(y_lst)
```

```
X_test=sc.fit_transform(x_test)
y_pred=predict(X_test)
```

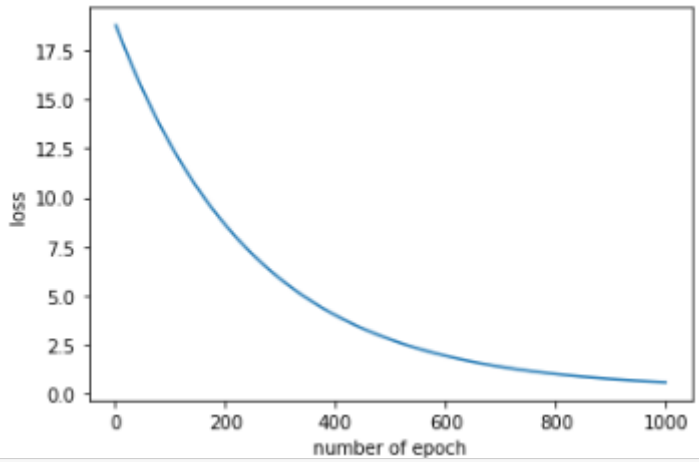
```
from sklearn.metrics import mean_squared_error, r2_score, mean_absolute_error
from math import sqrt
print("R2 score : %.3f" % r2_score(y_test, y_pred))
print("MSE: %.3f" % mean_squared_error(y_test, y_pred))
print("RMSE: %.3f" % sqrt(mean_squared_error(y_test, y_pred)))
print("MAE: %.3f" % sqrt(mean_absolute_error(y_test, y_pred)))
```

```
R2 score : 0.002
MSE: 0.229
RMSE: 0.479
MAE: 0.561
```

Fig: Gradient Plot for (1000,0.1) and calculation of errors

Here the gradient function (Loss or Cost) gradually decreases and after a point become constant and all kind of error values are calculated.

- (1000,0.001)




```
print("weight:",w)
print("bias:",b)
```

```
weight: [-0.11383408]
bias: 3.6250277086579943
```

```
def predict(inp):
    y_lst=[]
    for i in range(len(inp)):
        y_lst.append(w@inp[i]+b)
    return np.array(y_lst)
```

```
X_test=sc.fit_transform(x_test)
y_pred=predict(X_test)
```

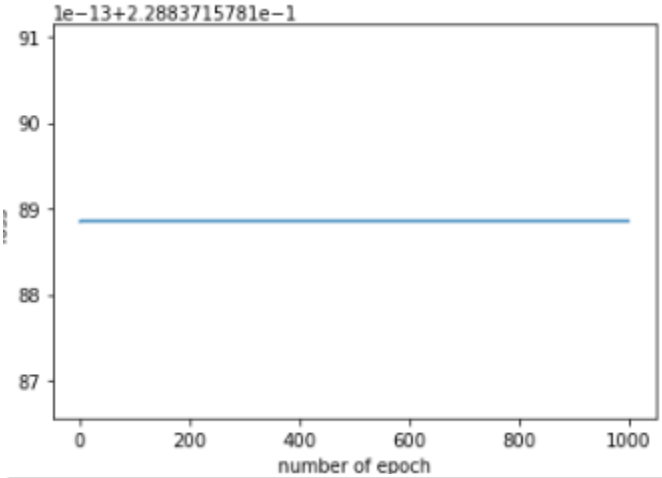
```
from sklearn.metrics import mean_squared_error, r2_score, mean_absolute_error
from math import sqrt
print("R2 score : %.3f" % r2_score(y_test, y_pred))
print("MSE: %.3f" % mean_squared_error(y_test, y_pred))
print("RMSE: %.3f" % sqrt(mean_squared_error(y_test, y_pred)))
print("MAE: %.3f" % sqrt(mean_absolute_error(y_test, y_pred)))
```

```
R2 score : -1.454
MSE: 0.564
RMSE: 0.751
MAE: 0.819
```

Fig: Gradient Plot for (1000,0.001) and calculation of errors

Here the gradient function (Loss or Cost) decreases in a curve shape and after a point become constant and all kind of error values are calculated.

- (1000,0.5)




```
print("weight:",w)
print("bias:",b)
```

```
weight: [0.02532093]
bias: 4.1910961022344475
```

```
def predict(inp):
    y_lst=[]
    for i in range(len(inp)):
        y_lst.append(w@inp[i]+b)
    return np.array(y_lst)
```

```
X_test=sc.fit_transform(x_test)
y_pred=predict(X_test)
```

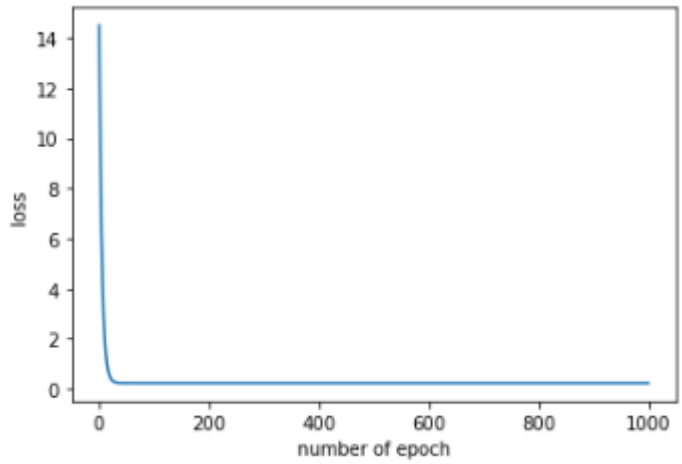
```
from sklearn.metrics import mean_squared_error, r2_score, mean_absolute_error
from math import sqrt
print("R2 score : %.3f" % r2_score(y_test, y_pred))
print("MSE: %.3f" % mean_squared_error(y_test, y_pred))
print("RMSE: %.3f" % sqrt(mean_squared_error(y_test, y_pred)))
print("MAE: %.3f" % sqrt(mean_absolute_error(y_test, y_pred)))
```

```
R2 score : 0.002
MSE: 0.229
RMSE: 0.479
MAE: 0.561
```

Fig: Gradient Plot for (1000,0.5) and calculation of errors

Here the gradient function (Loss or Cost) is constant for the whole duration and all kind of error values are calculated.

- (1000,0.05)



```
print("weight:",w)
print("bias:",b)
```

```
weight: [0.02532093]
bias: 4.191096102234443
```

```
def predict(inp):
    y_lst=[]
    for i in range(len(inp)):
        y_lst.append(w@inp[i]+b)
    return np.array(y_lst)
```

```
X_test=sc.fit_transform(x_test)
y_pred=predict(X_test)
```

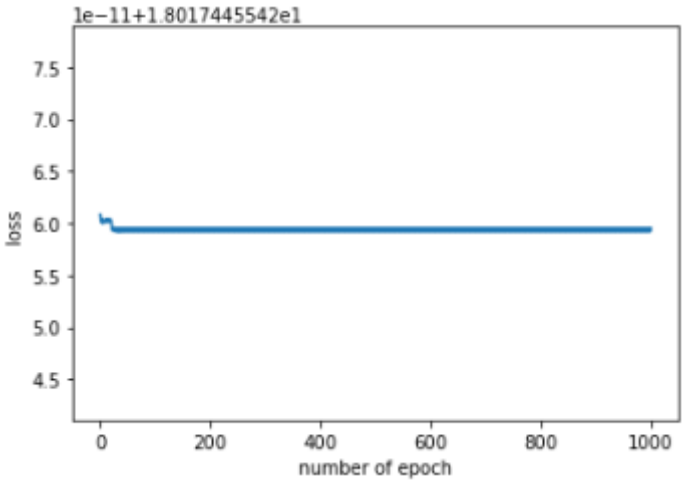
```
from sklearn.metrics import mean_squared_error, r2_score, mean_absolute_error
from math import sqrt
print("R2 score : %.3f" % r2_score(y_test, y_pred))
print("MSE: %.3f" % mean_squared_error(y_test, y_pred))
print("RMSE: %.3f" % sqrt(mean_squared_error(y_test, y_pred)))
print("MAE: %.3f" % sqrt(mean_absolute_error(y_test, y_pred)))
```

```
R2 score : 0.002
MSE: 0.229
RMSE: 0.479
MAE: 0.561
```

Fig: Gradient Plot for (1000,0.05) and calculation of errors

Here the gradient function (Loss or Cost) gradually decreases and after a point become constant and all kind of error values are calculated.

- (1000,1)




```
print("weight:",w)
print("bias:",b)
```

```
weight: [-0.44724848]
bias: 2.4513724383723456e-13
```

```
def predict(inp):
    y_lst=[]
    for i in range(len(inp)):
        y_lst.append(w@inp[i]+b)
    return np.array(y_lst)
```

```
X_test=sc.fit_transform(x_test)
y_pred=predict(X_test)
```

```
from sklearn.metrics import mean_squared_error, r2_score, mean_absolute_error
from math import sqrt
print("R2 score : %.3f" % r2_score(y_test, y_pred))
print("MSE: %.3f" % mean_squared_error(y_test, y_pred))
print("RMSE: %.3f" % sqrt(mean_squared_error(y_test, y_pred)))
print("MAE: %.3f" % sqrt(mean_absolute_error(y_test, y_pred)))
```

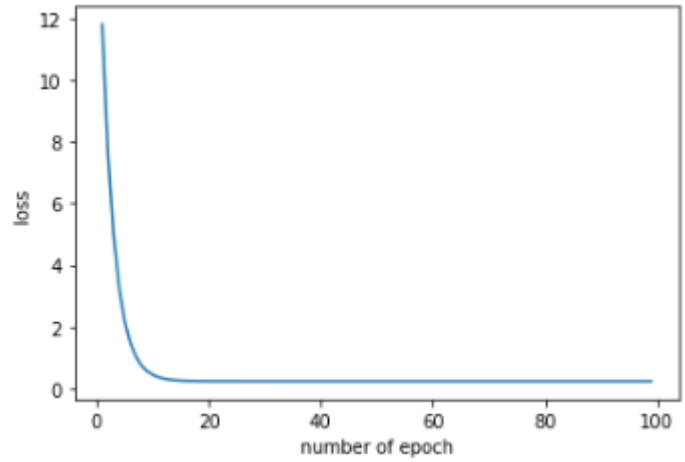
```
R2 score : -77.217
MSE: 17.982
RMSE: 4.241
MAE: 2.046
```

Fig: Gradient Plot for (1000,1) and calculation of errors

Here the gradient function (Loss or Cost) has little bit decreased in starting but after sometime it become and all kind of error values are calculated.

For Size

- (100,0.1)




```
print("weight:",w)
print("bias:",b)
```

```
weight: [0.04233722]
bias: 4.191096101380706
```

```
def predict(inp):
    y_lst=[]
    for i in range(len(inp)):
        y_lst.append(w@inp[i]+b)
    return np.array(y_lst)
```

```
X_test=sc.fit_transform(x_test)
y_pred=predict(X_test)
```

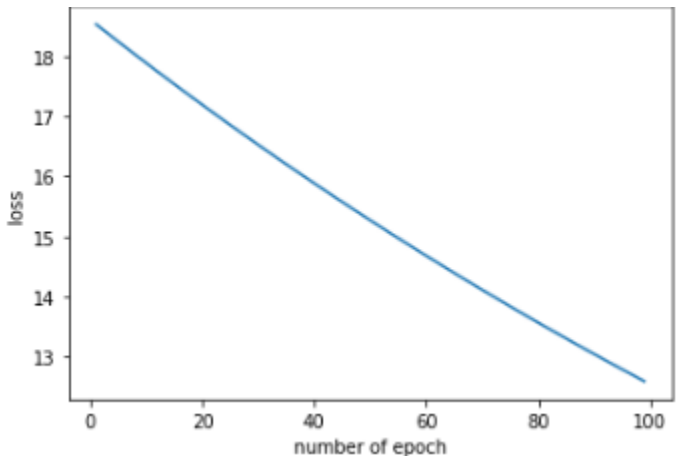
```
from sklearn.metrics import mean_squared_error, r2_score, mean_absolute_error
from math import sqrt
print("R2 score : %.3f" % r2_score(y_test, y_pred))
print("MSE: %.3f" % mean_squared_error(y_test, y_pred))
print("RMSE: %.3f" % sqrt(mean_squared_error(y_test, y_pred)))
print("MAE: %.3f" % sqrt(mean_absolute_error(y_test, y_pred)))
```

```
R2 score : 0.006
MSE: 0.228
RMSE: 0.478
MAE: 0.559
```

Fig: Gradient Plot for (100,0.1) and calculation of errors

Here the gradient function (Loss or Cost) gradually decreases and after a point become constant and all kind of error values are calculated.

- (100,0.001)



```
print("weight:",w)
print("bias:",b)
```

```
weight: [0.77920437]
bias: 0.7604039576862712
```

```
def predict(inp):
    y_lst=[]
    for i in range(len(inp)):
        y_lst.append(w@inp[i]+b)
    return np.array(y_lst)
```

```
X_test=sc.fit_transform(x_test)
y_pred=predict(X_test)
```

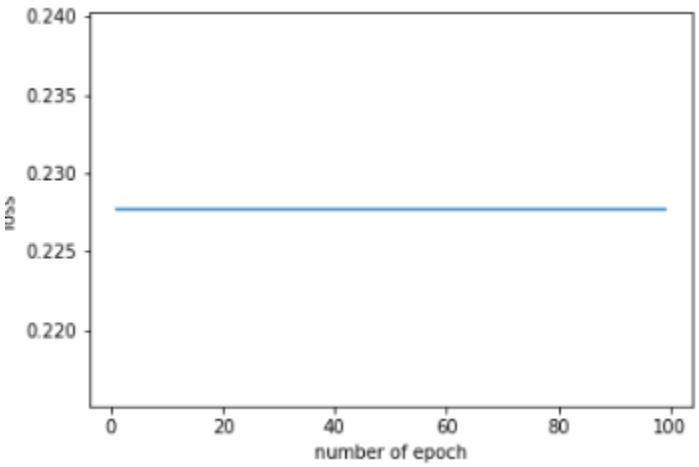
```
from sklearn.metrics import mean_squared_error, r2_score, mean_absolute_error
from math import sqrt
print("R2 score : %.3f" % r2_score(y_test, y_pred))
print("MSE: %.3f" % mean_squared_error(y_test, y_pred))
print("RMSE: %.3f" % sqrt(mean_squared_error(y_test, y_pred)))
print("MAE: %.3f" % sqrt(mean_absolute_error(y_test, y_pred)))
```

```
R2 score : -53.454
MSE: 12.519
RMSE: 3.538
MAE: 1.854
```

Fig: Gradient Plot for (100,0.001) and calculation of errors

Here the gradient function (Loss or Cost) decreases in a straight line path and does not become constant and all kind of error values are calculated.

- (100,0.5)




```
print("weight:",w)
print("bias:",b)
```

```
weight: [0.04233722]
bias: 4.1910961022344475
```

```
def predict(inp):
    y_lst=[]
    for i in range(len(inp)):
        y_lst.append(w@inp[i]+b)
    return np.array(y_lst)
```

```
X_test=sc.fit_transform(x_test)
y_pred=predict(X_test)
```

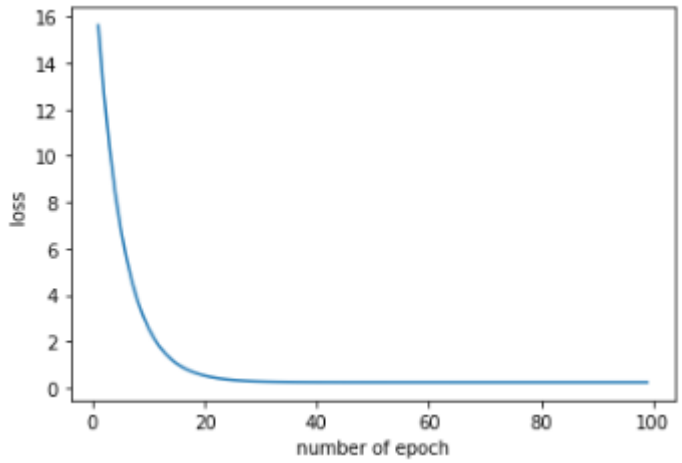
```
from sklearn.metrics import mean_squared_error, r2_score, mean_absolute_error
from math import sqrt
print("R2 score : %.3f" % r2_score(y_test, y_pred))
print("MSE: %.3f" % mean_squared_error(y_test, y_pred))
print("RMSE: %.3f" % sqrt(mean_squared_error(y_test, y_pred)))
print("MAE: %.3f" % sqrt(mean_absolute_error(y_test, y_pred)))
```

```
R2 score : 0.006
MSE: 0.228
RMSE: 0.478
MAE: 0.559
```

Fig: Gradient Plot for (100,0.5) and calculation of errors

Here the gradient function (Loss or Cost) is constant for the entire duration and all kind of error values are calculated.

- (100,0.05)




```
print("weight:",w)
print("bias:",b)
```

```
weight: [0.04230524]
bias: 4.190984780859099
```

```
def predict(inp):
    y_lst=[]
    for i in range(len(inp)):
        y_lst.append(w@inp[i]+b)
    return np.array(y_lst)
```

```
X_test=sc.fit_transform(x_test)
y_pred=predict(X_test)
```

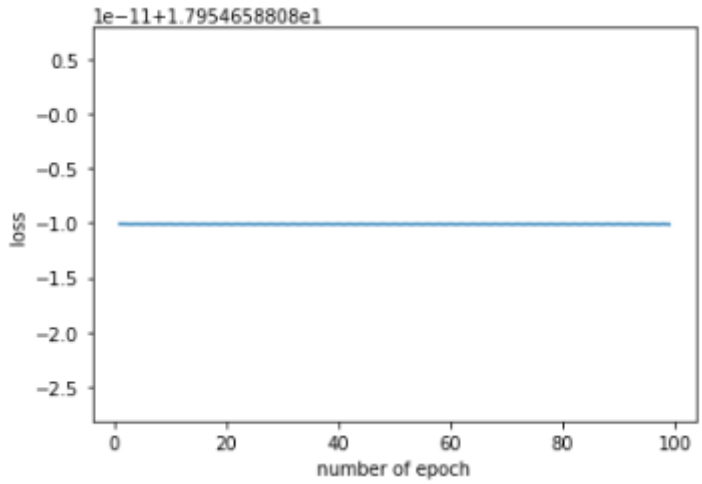
```
from sklearn.metrics import mean_squared_error, r2_score, mean_absolute_error
from math import sqrt
print("R2 score : %.3f" % r2_score(y_test, y_pred))
print("MSE: %.3f" % mean_squared_error(y_test, y_pred))
print("RMSE: %.3f" % sqrt(mean_squared_error(y_test, y_pred)))
print("MAE: %.3f" % sqrt(mean_absolute_error(y_test, y_pred)))
```

```
R2 score : 0.006
MSE: 0.228
RMSE: 0.478
MAE: 0.559
```

Fig: Gradient Plot for (100,0.05) and calculation of errors

Here the gradient function (Loss or Cost) gradually decreases and after a point become constant and all kind of error values are calculated.

- (100, 1)



```
print("weight:",w)
print("bias:",b)
```

```
weight: [0.44443969]
bias: 1.2434497875801753e-14
```

```
def predict(inp):
    y_lst=[]
    for i in range(len(inp)):
        y_lst.append(w@inp[i]+b)
    return np.array(y_lst)
```

```
X_test=sc.fit_transform(x_test)
y_pred=predict(X_test)
```

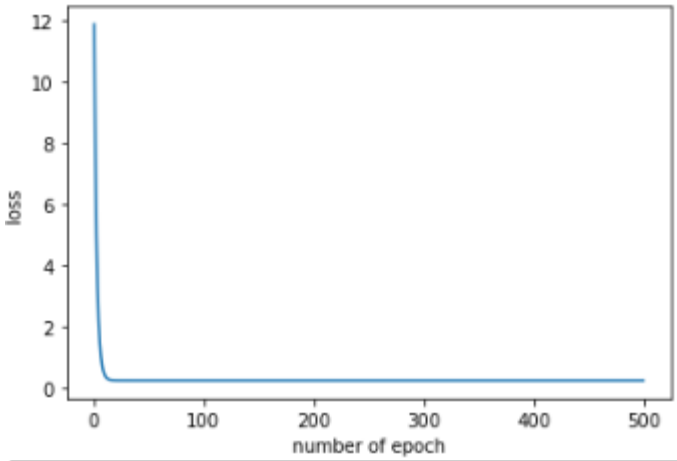
```
from sklearn.metrics import mean_squared_error, r2_score, mean_absolute_error
from math import sqrt
print("R2 score : %.3f" % r2_score(y_test, y_pred))
print("MSE: %.3f" % mean_squared_error(y_test, y_pred))
print("RMSE: %.3f" % sqrt(mean_squared_error(y_test, y_pred)))
print("MAE: %.3f" % sqrt(mean_absolute_error(y_test, y_pred)))
```

```
R2 score : -76.966
MSE: 17.925
RMSE: 4.234
MAE: 2.046
```

Fig: Gradient Plot for (100,1) and calculation of errors

Here the gradient function (Loss or Cost) is constant for the whole time and all kind of error values are calculated.

- (500,0.1)




```
print("weight:",w)
print("bias:",b)
```

```
weight: [0.04233722]
bias: 4.191096102234446
```

```
def predict(inp):
    y_lst=[]
    for i in range(len(inp)):
        y_lst.append(w@inp[i]+b)
    return np.array(y_lst)
```

```
X_test=sc.fit_transform(x_test)
y_pred=predict(X_test)
```

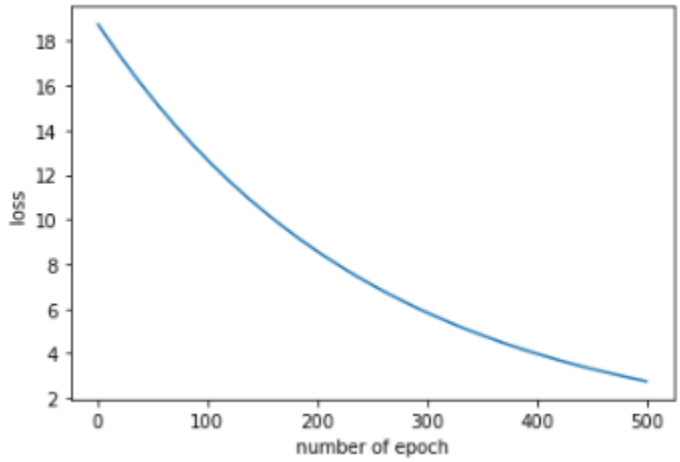
```
from sklearn.metrics import mean_squared_error, r2_score, mean_absolute_error
from math import sqrt
print("R2 score : %.3f" % r2_score(y_test, y_pred))
print("MSE: %.3f" % mean_squared_error(y_test, y_pred))
print("RMSE: %.3f" % sqrt(mean_squared_error(y_test, y_pred)))
print("MAE: %.3f" % sqrt(mean_absolute_error(y_test, y_pred)))
```

```
R2 score : 0.006
MSE: 0.228
RMSE: 0.478
MAE: 0.559
```

Fig: Gradient Plot for (500,0.1) and calculation of errors

Here the gradient function (Loss or Cost) gradually decreases and after a point become constant and all kind of error values are calculated.

- (500,0.001)




```
print("weight:",w)
print("bias:",b)
```

```
weight: [-0.32898246]
bias: 2.650821114475316
```

```
def predict(inp):
    y_lst=[]
    for i in range(len(inp)):
        y_lst.append(w@inp[i]+b)
    return np.array(y_lst)
```

```
X_test=sc.fit_transform(x_test)
y_pred=predict(X_test)
```

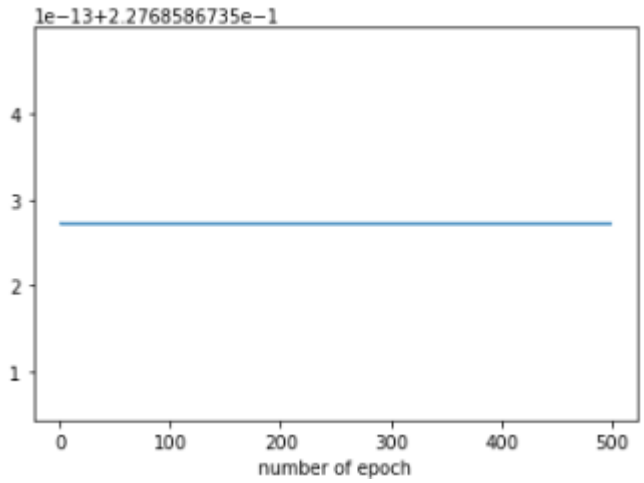
```
from sklearn.metrics import mean_squared_error, r2_score, mean_absolute_error
from math import sqrt
print("R2 score : %.3f" % r2_score(y_test, y_pred))
print("MSE: %.3f" % mean_squared_error(y_test, y_pred))
print("RMSE: %.3f" % sqrt(mean_squared_error(y_test, y_pred)))
print("MAE: %.3f" % sqrt(mean_absolute_error(y_test, y_pred)))
```

```
R2 score : -10.844
MSE: 2.723
RMSE: 1.650
MAE: 1.249
```

Fig: Gradient Plot for (500,0.001) and calculation of errors

Here the gradient function (Loss or Cost) decreases in a curved shape and after a point become constant and all kind of error values are calculated.

- (500,0.5)



```
print("weight:",w)
print("bias:",b)
```

```
weight: [0.04233722]
bias: 4.1910961022344475
```

```
def predict(inp):
    y_lst=[]
    for i in range(len(inp)):
        y_lst.append(w@inp[i]+b)
    return np.array(y_lst)
```

```
X_test=sc.fit_transform(x_test)
y_pred=predict(X_test)
```

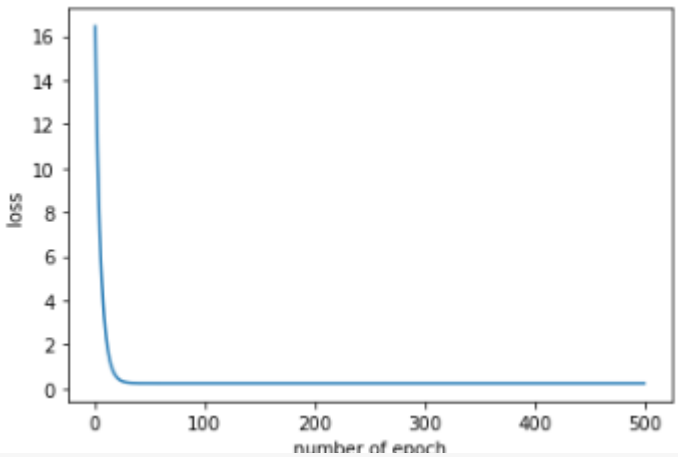
```
from sklearn.metrics import mean_squared_error, r2_score, mean_absolute_error
from math import sqrt
print("R2 score : %.3f" % r2_score(y_test, y_pred))
print("MSE: %.3f" % mean_squared_error(y_test, y_pred))
print("RMSE: %.3f" % sqrt(mean_squared_error(y_test, y_pred)))
print("MAE: %.3f" % sqrt(mean_absolute_error(y_test, y_pred)))
```

```
R2 score : 0.006
MSE: 0.228
RMSE: 0.478
MAE: 0.559
```

Fig: Gradient Plot for (500,0.5) and calculation of errors

Here the gradient function (Loss or Cost) is constant for the whole duration and all kind of error values are calculated.

- (500,0.05)




```
print("weight:",w)
print("bias:",b)
```

```
weight: [0.04233722]
bias: 4.191096102234443
```

```
def predict(inp):
    y_lst=[]
    for i in range(len(inp)):
        y_lst.append(w@inp[i]+b)
    return np.array(y_lst)
```

```
X_test=sc.fit_transform(x_test)
y_pred=predict(X_test)
```

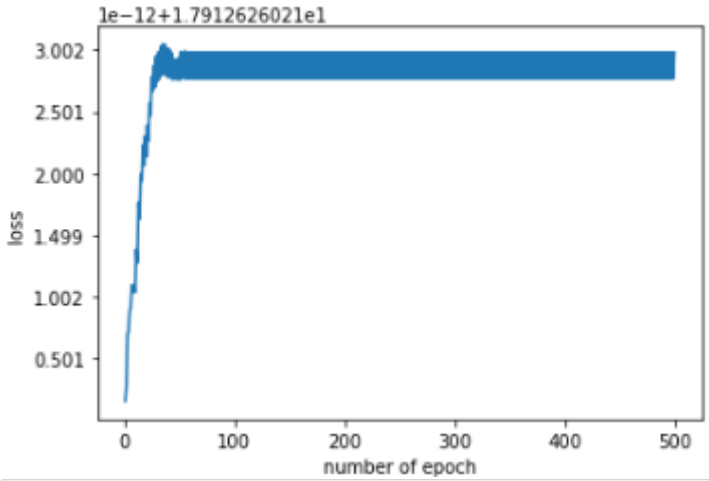
```
from sklearn.metrics import mean_squared_error, r2_score, mean_absolute_error
from math import sqrt
print("R2 score : %.3f" % r2_score(y_test, y_pred))
print("MSE: %.3f" % mean_squared_error(y_test, y_pred))
print("RMSE: %.3f" % sqrt(mean_squared_error(y_test, y_pred)))
print("MAE: %.3f" % sqrt(mean_absolute_error(y_test, y_pred)))
```

```
R2 score : 0.006
MSE: 0.228
RMSE: 0.478
MAE: 0.559
```

Fig: Gradient Plot for (500,0.05) and calculation of errors

Here the gradient function (Loss or Cost) gradually decreases and after a point become constant and all kind of error values are calculated.

- (500,1)




```
print("weight:",w)
print("bias:",b)
```

```
weight: [0.38824705]
bias: -3.5349501104064984e-13
```

```
def predict(inp):
    y_lst=[]
    for i in range(len(inp)):
        y_lst.append(w@inp[i]+b)
    return np.array(y_lst)
```

```
X_test=sc.fit_transform(x_test)
y_pred=predict(X_test)
```

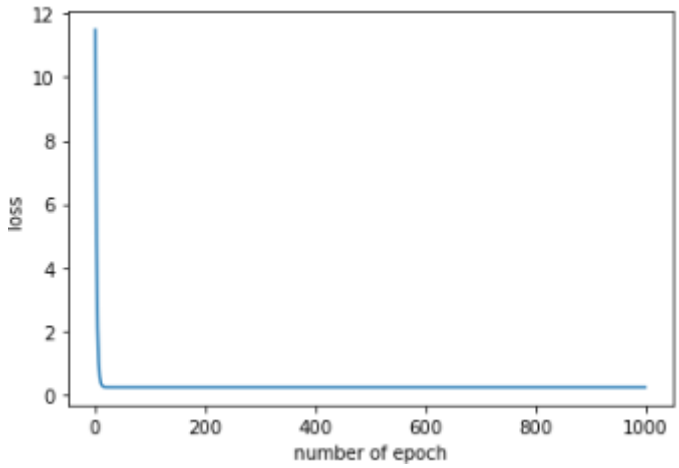
```
from sklearn.metrics import mean_squared_error, r2_score, mean_absolute_error
from math import sqrt
print("R2 score : %.3f" % r2_score(y_test, y_pred))
print("MSE: %.3f" % mean_squared_error(y_test, y_pred))
print("RMSE: %.3f" % sqrt(mean_squared_error(y_test, y_pred)))
print("MAE: %.3f" % sqrt(mean_absolute_error(y_test, y_pred)))
```

```
R2 score : -76.781
MSE: 17.882
RMSE: 4.229
MAE: 2.046
```

Fig: Gradient Plot for (500,1) and calculation of errors

Here the gradient function (Loss or Cost) gradually increases in starting and after reaching maxima fluctuates for some duration and then become constant and all kind of error values are calculated.

- (1000,0.1)



```
print("weight:",w)
print("bias:",b)
```

```
weight: [0.04233722]
bias: 4.191096102234446
```

```
def predict(inp):
    y_lst=[]
    for i in range(len(inp)):
        y_lst.append(w@inp[i]+b)
    return np.array(y_lst)
```

```
X_test=sc.fit_transform(x_test)
y_pred=predict(X_test)
```

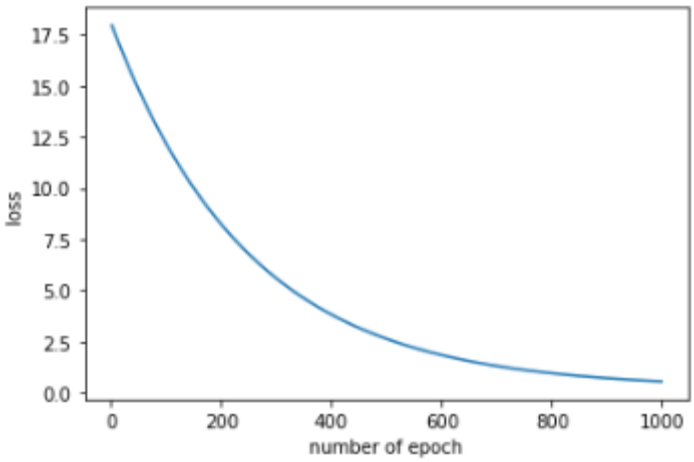
```
from sklearn.metrics import mean_squared_error, r2_score, mean_absolute_error
from math import sqrt
print("R2 score : %.3f" % r2_score(y_test, y_pred))
print("MSE: %.3f" % mean_squared_error(y_test, y_pred))
print("RMSE: %.3f" % sqrt(mean_squared_error(y_test, y_pred)))
print("MAE: %.3f" % sqrt(mean_absolute_error(y_test, y_pred)))
```

```
R2 score : 0.006
MSE: 0.228
RMSE: 0.478
MAE: 0.559
```

Fig: Gradient Plot for (1000,0.1) and calculation of errors

Here the gradient function (Loss or Cost) gradually decreases and after a point become constant and all kind of error values are calculated.

- (1000,0.001)




```
print("weight:",w)
print("bias:",b)
```

```
weight: [-0.02194646]
bias: 3.6250277086579947
```

```
def predict(inp):
    y_lst=[]
    for i in range(len(inp)):
        y_lst.append(w@inp[i]+b)
    return np.array(y_lst)
```

```
X_test=sc.fit_transform(x_test)
y_pred=predict(X_test)
```

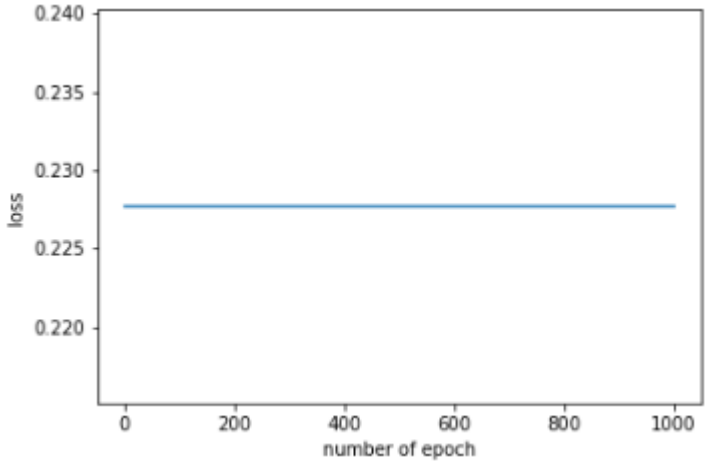
```
from sklearn.metrics import mean_squared_error, r2_score, mean_absolute_error
from math import sqrt
print("R2 score : %.3f" % r2_score(y_test, y_pred))
print("MSE: %.3f" % mean_squared_error(y_test, y_pred))
print("RMSE: %.3f" % sqrt(mean_squared_error(y_test, y_pred)))
print("MAE: %.3f" % sqrt(mean_absolute_error(y_test, y_pred)))
```

```
R2 score : -1.383
MSE: 0.548
RMSE: 0.740
MAE: 0.817
```

Fig: Gradient Plot for (1000,0.001) and calculation of errors

Here the gradient function (Loss or Cost) decreases in a curve path and after a point become constant and all kind of error values are calculated.

- (1000,0.5)




```
print("weight:",w)
print("bias:",b)
```

```
weight: [0.04233722]
bias: 4.1910961022344475
```

```
def predict(inp):
    y_lst=[]
    for i in range(len(inp)):
        y_lst.append(w@inp[i]+b)
    return np.array(y_lst)
```

```
X_test=sc.fit_transform(x_test)
y_pred=predict(X_test)
```

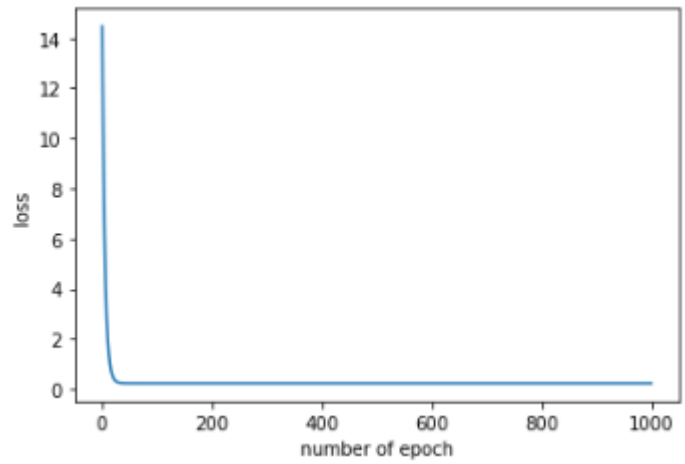
```
from sklearn.metrics import mean_squared_error, r2_score, mean_absolute_error
from math import sqrt
print("R2 score : %.3f" % r2_score(y_test, y_pred))
print("MSE: %.3f" % mean_squared_error(y_test, y_pred))
print("RMSE: %.3f" % sqrt(mean_squared_error(y_test, y_pred)))
print("MAE: %.3f" % sqrt(mean_absolute_error(y_test, y_pred)))
```

```
R2 score : 0.006
MSE: 0.228
RMSE: 0.478
MAE: 0.559
```

Fig: Gradient Plot for (1000,0.5) and calculation of errors

Here the gradient function (Loss or Cost) is constant for the whole duration and all kind of error values are calculated.

- (1000,0.05)



```
print("weight:",w)
print("bias:",b)
```

```
weight: [0.04233722]
bias: 4.191096102234443
```

```
def predict(inp):
    y_lst=[]
    for i in range(len(inp)):
        y_lst.append(w@inp[i]+b)
    return np.array(y_lst)
```

```
X_test=sc.fit_transform(x_test)
y_pred=predict(X_test)
```

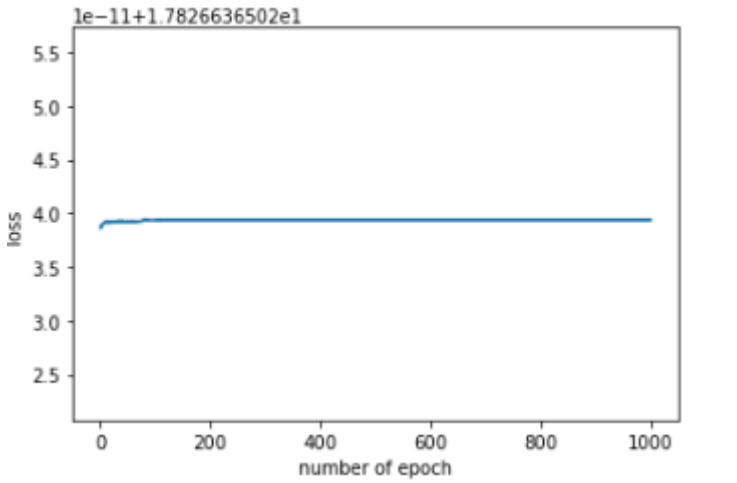
```
from sklearn.metrics import mean_squared_error, r2_score, mean_absolute_error
from math import sqrt
print("R2 score : %.3f" % r2_score(y_test, y_pred))
print("MSE: %.3f" % mean_squared_error(y_test, y_pred))
print("RMSE: %.3f" % sqrt(mean_squared_error(y_test, y_pred)))
print("MAE: %.3f" % sqrt(mean_absolute_error(y_test, y_pred)))
```

```
R2 score : 0.006
MSE: 0.228
RMSE: 0.478
MAE: 0.559
```

Fig: Gradient Plot for (1000,0.05) and calculation of errors

Here the gradient function (Loss or Cost) gradually decreases and after a point become constant and all kind of error values are calculated.

- (1000,1)




```
print("weight:",w)
print("bias:",b)
```

```
weight: [0.225815]
bias: -1.1013412404281553e-13
```

```
def predict(inp):
    y_lst=[]
    for i in range(len(inp)):
        y_lst.append(w@inp[i]+b)
    return np.array(y_lst)
```

```
X_test=sc.fit_transform(x_test)
y_pred=predict(X_test)
```

```
from sklearn.metrics import mean_squared_error, r2_score, mean_absolute_error
from math import sqrt
print("R2 score : %.3f" % r2_score(y_test, y_pred))
print("MSE: %.3f" % mean_squared_error(y_test, y_pred))
print("RMSE: %.3f" % sqrt(mean_squared_error(y_test, y_pred)))
print("MAE: %.3f" % sqrt(mean_absolute_error(y_test, y_pred)))
```

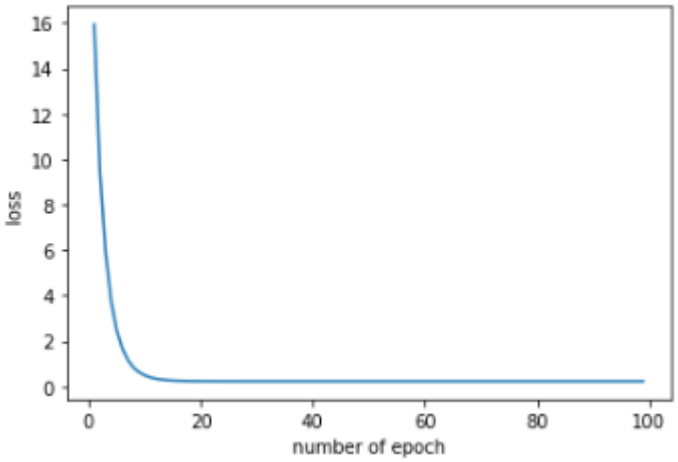
```
R2 score : -76.401
MSE: 17.795
RMSE: 4.218
MAE: 2.046
```

Fig: Gradient Plot for (1000,1) and calculation of errors

Here the gradient function (Loss or Cost) slightly increases and after a point become constant for the whole duration and all kind of error values are calculated.

For Multiple Regression (All Predictors)

- (100,0.1)




```
print("weight:",w)
print("bias:",b)
```

```
weight: [ 0.02030842  0.03123877  0.00581486  0.02827815 -0.01317887  0.00128937
 -0.01548869]
bias: 4.191096101380706
```

```
def predict(inp):
    y_lst=[]
    for i in range(len(inp)):
        y_lst.append(w@inp[i]+b)
    return np.array(y_lst)
```

```
X_test=sc.fit_transform(x_test)
y_pred=predict(X_test)
```

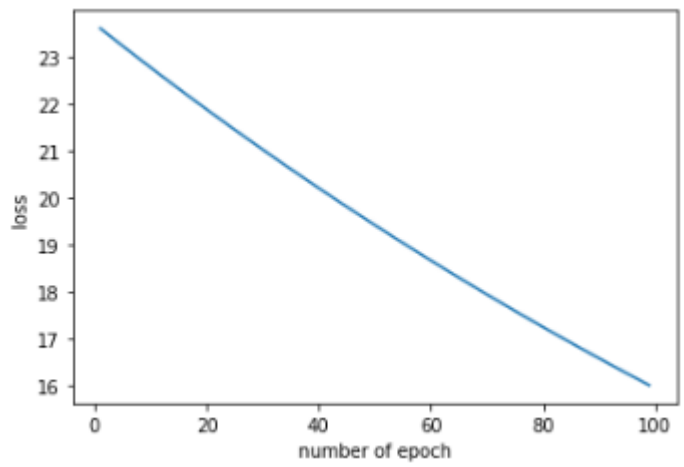
```
from sklearn.metrics import mean_squared_error, r2_score, mean_absolute_error
from math import sqrt
print("R2 score : %.3f" % r2_score(y_test, y_pred))
print("MSE: %.3f" % mean_squared_error(y_test, y_pred))
print("RMSE: %.3f" % sqrt(mean_squared_error(y_test, y_pred)))
print("MAE: %.3f" % sqrt(mean_absolute_error(y_test, y_pred)))
```

```
R2 score : 0.013
MSE: 0.227
RMSE: 0.476
MAE: 0.555
```

Fig: Gradient Plot for (100,0.1) and calculation of errors

Here the gradient function (Loss or Cost) gradually decreases and after a point become constant and all kind of error values are calculated.

- (100,0.001)



```
print("weight:",w)
print("bias:",b)
```

```
weight: [ 0.99064405 -0.7341565   0.32170121 -0.24817128 -0.92418739 -0.56305897
 -1.31767553]
bias: 0.7604039576862712
```

```
def predict(inp):
    y_lst=[]
    for i in range(len(inp)):
        y_lst.append(w@inp[i]+b)
    return np.array(y_lst)
```

```
X_test=sc.fit_transform(x_test)
y_pred=predict(X_test)
```

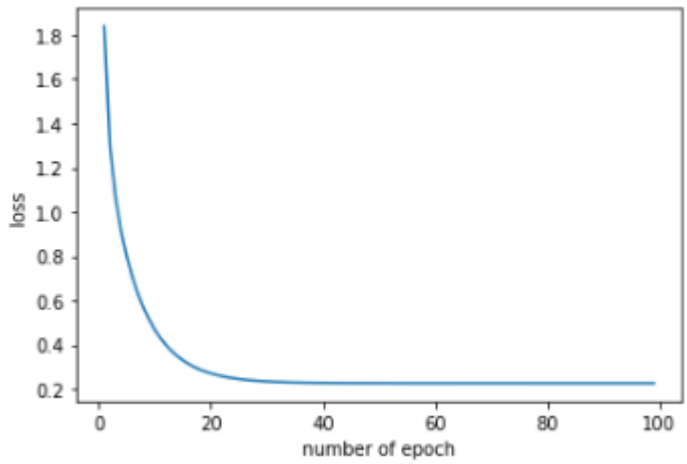
```
from sklearn.metrics import mean_squared_error, r2_score, mean_absolute_error
from math import sqrt
print("R2 score : %.3f" % r2_score(y_test, y_pred))
print("MSE: %.3f" % mean_squared_error(y_test, y_pred))
print("RMSE: %.3f" % sqrt(mean_squared_error(y_test, y_pred)))
print("MAE: %.3f" % sqrt(mean_absolute_error(y_test, y_pred)))
```

```
R2 score : -68.255
MSE: 15.922
RMSE: 3.990
MAE: 1.879
```

Fig: Gradient Plot for (100,0.001) and calculation of errors

Here the gradient function (Loss or Cost) decreases in a straight-line path and does not become constant and all kind of error values are calculated.

- (100,0.5)




```
print("weight:",w)
print("bias:",b)
```

```
weight: [ 0.02038335  0.03133318  0.00593962  0.02826161 -0.01318756  0.00132339
 -0.01550335]
bias: 4.1910961022344475
```

```
def predict(inp):
    y_lst=[]
    for i in range(len(inp)):
        y_lst.append(w@inp[i]+b)
    return np.array(y_lst)
```

```
X_test=sc.fit_transform(x_test)
y_pred=predict(X_test)
```

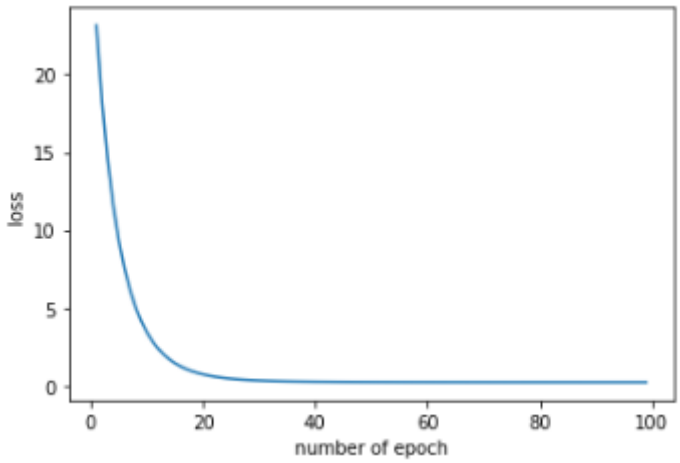
```
from sklearn.metrics import mean_squared_error, r2_score, mean_absolute_error
from math import sqrt
print("R2 score : %.3f" % r2_score(y_test, y_pred))
print("MSE: %.3f" % mean_squared_error(y_test, y_pred))
print("RMSE: %.3f" % sqrt(mean_squared_error(y_test, y_pred)))
print("MAE: %.3f" % sqrt(mean_absolute_error(y_test, y_pred)))
```

```
R2 score : 0.012
MSE: 0.227
RMSE: 0.476
MAE: 0.555
```

Fig: Gradient Plot for (100,0.5) and calculation of errors

Here the gradient function (Loss or Cost) gradually decreases and after a point become constant and all kind of error values are calculated.

- (100,0.05)




```
print("weight:",w)
print("bias:",b)
```

```
weight: [ 0.04075236  0.02436096 -0.01091987  0.02792411 -0.0131578   0.00246345
 -0.01661751]
bias: 4.190984780859099
```

```
def predict(inp):
    y_lst=[]
    for i in range(len(inp)):
        y_lst.append(w@inp[i]+b)
    return np.array(y_lst)
```

```
X_test=sc.fit_transform(x_test)
y_pred=predict(X_test)
```

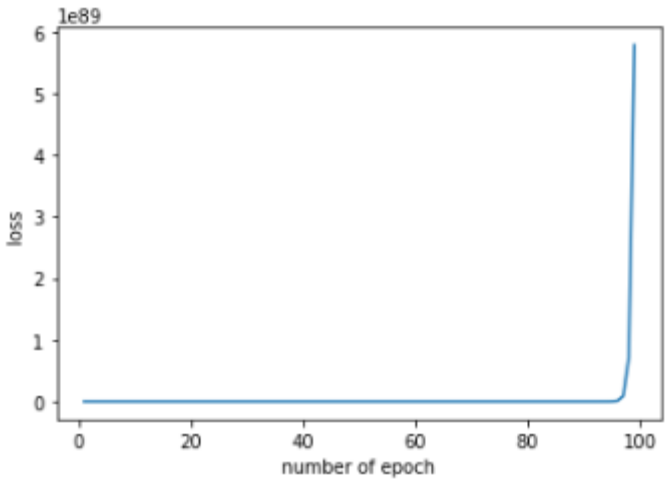
```
from sklearn.metrics import mean_squared_error, r2_score, mean_absolute_error
from math import sqrt
print("R2 score : %.3f" % r2_score(y_test, y_pred))
print("MSE: %.3f" % mean_squared_error(y_test, y_pred))
print("RMSE: %.3f" % sqrt(mean_squared_error(y_test, y_pred)))
print("MAE: %.3f" % sqrt(mean_absolute_error(y_test, y_pred)))
```

```
R2 score : 0.011
MSE: 0.227
RMSE: 0.477
MAE: 0.555
```

Fig: Gradient Plot for (100,0.05) and calculation of errors

Here the gradient function (Loss or Cost) gradually decreases and after a point become constant and all kind of error values are calculated.

- (100, 1)



```
print("weight:",w)
print("bias:",b)
```

```
weight: [-9.52313264e+44 -7.66203695e+44 -8.83586540e+44  1.52934407e+44
  7.84025573e+43 -3.18087465e+44  1.42977417e+44]
bias: -2.9375236313602316e+29
```

```
def predict(inp):
    y_lst=[]
    for i in range(len(inp)):
        y_lst.append(w@inp[i]+b)
    return np.array(y_lst)
```

```
X_test=sc.fit_transform(x_test)
y_pred=predict(X_test)
```

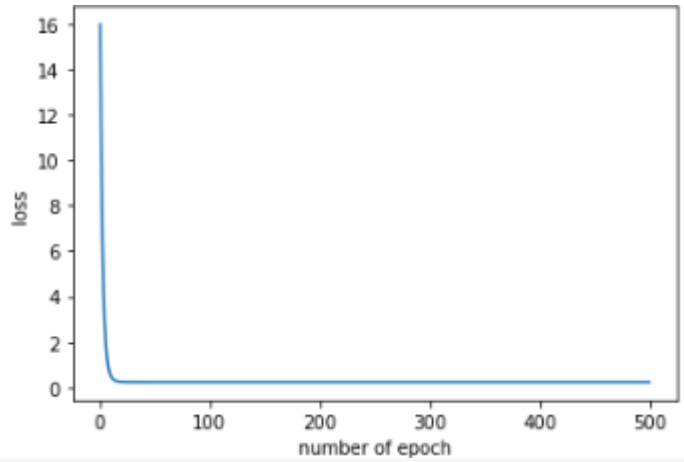
```
from sklearn.metrics import mean_squared_error, r2_score, mean_absolute_error
from math import sqrt
print("R2 score : %.3f" % r2_score(y_test, y_pred))
print("MSE: %.3f" % mean_squared_error(y_test, y_pred))
print("RMSE: %.3f" % sqrt(mean_squared_error(y_test, y_pred)))
print("MAE: %.3f" % sqrt(mean_absolute_error(y_test, y_pred)))
```

```
R2 score : -215306458974564273617154446696883597305173170295922903275403126232903417186024544441414445120.000
MSE: 4949971756646950898270240703521046358372101515442049345978074005117881881152008600023990272.000
RMSE: 22248531988980645171349683433918298755075932l6.000
MAE: 3240041220511036656844B.000
```

Fig: Gradient Plot for (100,1) and calculation of errors

Here the gradient function (Loss or Cost) is constant in the beginning and after 90 iterations it starts increasing gradually and all kind of error values are calculated.

- (500,0.1)




```
print("weight:",w)
print("bias:",b)
```

```
weight: [ 0.02027734  0.03124788  0.00584125  0.02827863 -0.01317883  0.00128798
 -0.01548743]
bias: 4.191096102234446
```

```
def predict(inp):
    y_lst=[]
    for i in range(len(inp)):
        y_lst.append(w@inp[i]+b)
    return np.array(y_lst)
```

```
X_test=sc.fit_transform(x_test)
y_pred=predict(X_test)
```

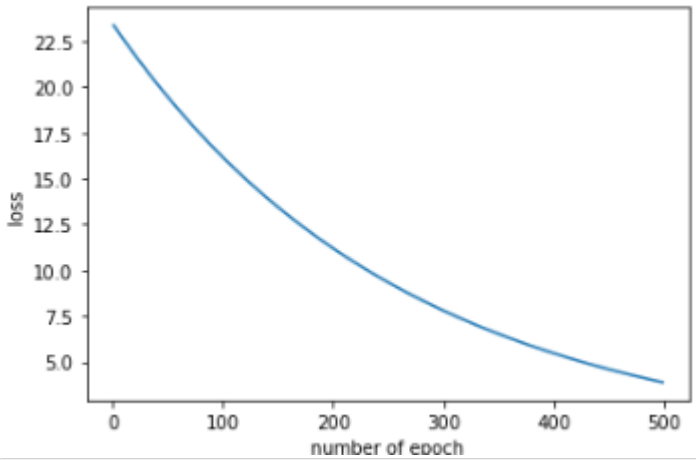
```
from sklearn.metrics import mean_squared_error, r2_score, mean_absolute_error
from math import sqrt
print("R2 score : %.3f" % r2_score(y_test, y_pred))
print("MSE: %.3f" % mean_squared_error(y_test, y_pred))
print("RMSE: %.3f" % sqrt(mean_squared_error(y_test, y_pred)))
print("MAE: %.3f" % sqrt(mean_absolute_error(y_test, y_pred)))
```

```
R2 score : 0.013
MSE: 0.227
RMSE: 0.476
MAE: 0.555
```

Fig: Gradient Plot for (500,0.1) and calculation of errors

Here the gradient function (Loss or Cost) gradually decreases and after a point become constant and all kind of error values are calculated.

- (500,0.001)




```
print("weight:",w)
print("bias:",b)
```

```
weight: [-0.48244206  0.74280934  0.18878042  0.69787656 -0.62735187 -0.51839715
  0.03377705]
bias: 2.650821114475315
```

```
def predict(inp):
    y_lst=[]
    for i in range(len(inp)):
        y_lst.append(w@inp[i]+b)
    return np.array(y_lst)
```

```
X_test=sc.fit_transform(x_test)
y_pred=predict(X_test)
```

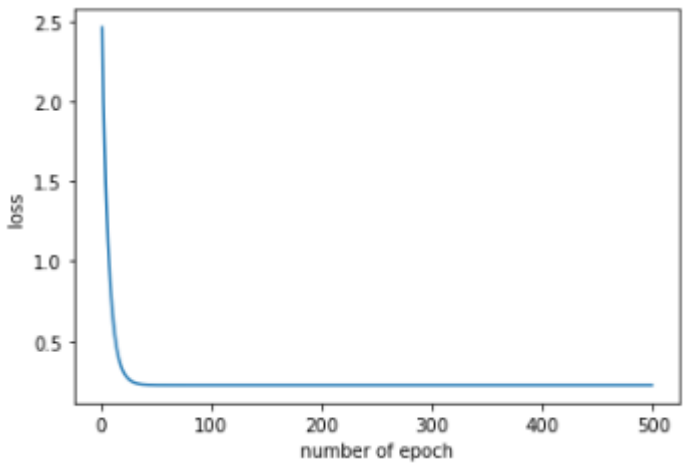
```
from sklearn.metrics import mean_squared_error, r2_score, mean_absolute_error
from math import sqrt
print("R2 score : %.3f" % r2_score(y_test, y_pred))
print("MSE: %.3f" % mean_squared_error(y_test, y_pred))
print("RMSE: %.3f" % sqrt(mean_squared_error(y_test, y_pred)))
print("MAE: %.3f" % sqrt(mean_absolute_error(y_test, y_pred)))
```

```
R2 score : -15.603
MSE: 3.817
RMSE: 1.954
MAE: 1.305
```

Fig: Gradient Plot for (500,0.001) and calculation of errors

Here the gradient function (Loss or Cost) decreases in a curve shape and after a point become constant and all kind of error values are calculated.

- (500,0.5)



```
print("weight:",w)
print("bias:",b)
```

```
weight: [ 0.02027734  0.03124788  0.00584125  0.02827863 -0.01317883  0.00128798
 -0.01548743]
bias: 4.1910961022344475
```

```
def predict(inp):
    y_lst=[]
    for i in range(len(inp)):
        y_lst.append(w@inp[i]+b)
    return np.array(y_lst)
```

```
X_test=sc.fit_transform(x_test)
y_pred=predict(X_test)
```

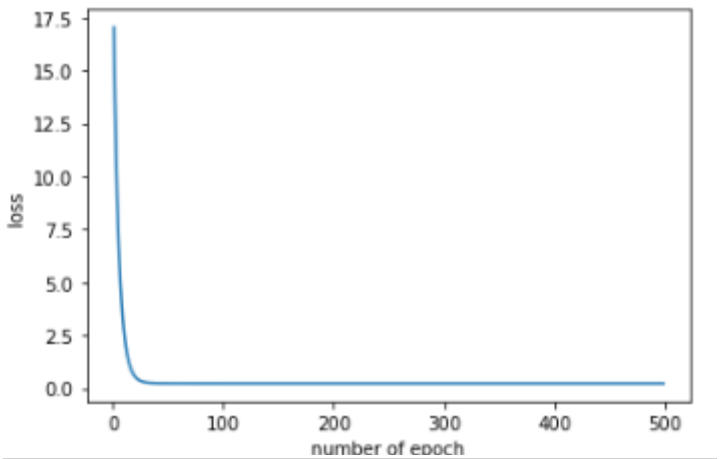
```
from sklearn.metrics import mean_squared_error, r2_score, mean_absolute_error
from math import sqrt
print("R2 score : %.3f" % r2_score(y_test, y_pred))
print("MSE: %.3f" % mean_squared_error(y_test, y_pred))
print("RMSE: %.3f" % sqrt(mean_squared_error(y_test, y_pred)))
print("MAE: %.3f" % sqrt(mean_absolute_error(y_test, y_pred)))
```

```
R2 score : 0.013
MSE: 0.227
RMSE: 0.476
MAE: 0.555
```

Fig: Gradient Plot for (500,0.5) and calculation of errors

Here the gradient function (Loss or Cost) gradually decreases and after a point become constant and all kind of error values are calculated.

- (500,0.05)




```
print("weight:",w)
print("bias:",b)
```

```
weight: [ 0.02027734  0.03124788  0.00584125  0.02827863 -0.01317883  0.00128798
 -0.01548743]
bias: 4.191096102234443
```

```
def predict(inp):
    y_lst=[]
    for i in range(len(inp)):
        y_lst.append(w@inp[i]+b)
    return np.array(y_lst)
```

```
X_test=sc.fit_transform(x_test)
y_pred=predict(X_test)
```

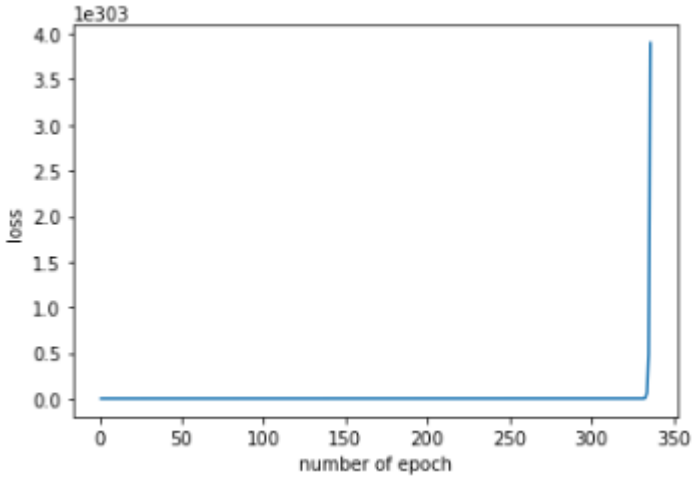
```
from sklearn.metrics import mean_squared_error, r2_score, mean_absolute_error
from math import sqrt
print("R2 score : %.3f" % r2_score(y_test, y_pred))
print("MSE: %.3f" % mean_squared_error(y_test, y_pred))
print("RMSE: %.3f" % sqrt(mean_squared_error(y_test, y_pred)))
print("MAE: %.3f" % sqrt(mean_absolute_error(y_test, y_pred)))
```

```
R2 score : 0.013
MSE: 0.227
RMSE: 0.476
MAE: 0.555
```

Fig: Gradient Plot for (500,0.05) and calculation of errors

Here the gradient function (Loss or Cost) gradually decreases and after a point become constant and all kind of error values are calculated.

- (500,1)




```
print("weight:",w)
print("bias:",b)
```

```
weight: [-5.28132391e+225 -4.24920039e+225 -4.90018032e+225  8.48141228e+224
  4.34803668e+224 -1.76404445e+225  7.92921910e+224]
bias: -1.3465536539794395e+210
```

```
def predict(inp):
    y_lst=[]
    for i in range(len(inp)):
        y_lst.append(w@inp[i]+b)
    return np.array(y_lst)
```

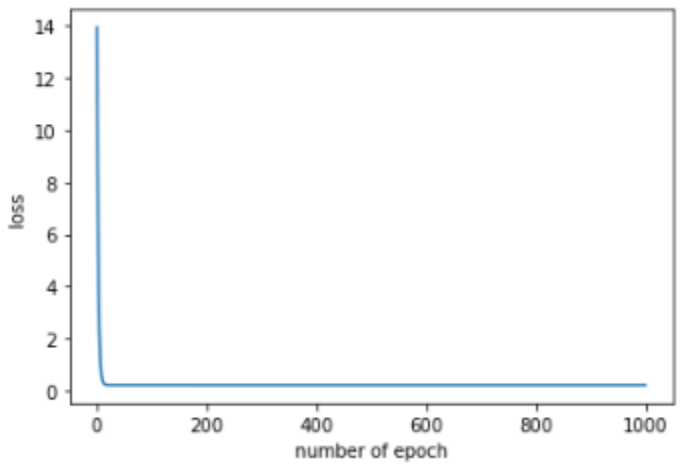
```
X_test=sc.fit_transform(x_test)
y_pred=predict(X_test)
```

```
from sklearn.metrics import mean_squared_error, r2_score, mean_absolute_error
from math import sqrt
print("R2 score : %.3f" % r2_score(y_test, y_pred))
print("MSE: %.3f" % mean_squared_error(y_test, y_pred))
print("RMSE: %.3f" % sqrt(mean_squared_error(y_test, y_pred)))
print("MAE: %.3f" % sqrt(mean_absolute_error(y_test, y_pred)))
```

Fig: Gradient Plot for (500,1) and calculation of errors

Here the gradient function (Loss or Cost) is constant in the beginning and after 300 iterations it starts increasing gradually and all kind of error values are calculated.

- (1000,0.1)



```
print("weight:",w)
print("bias:",b)
```

```
weight: [ 0.02027734  0.03124788  0.00584125  0.02827863 -0.01317883  0.00128798
 -0.01548743]
bias: 4.191096102234446
```

```
def predict(inp):
    y_lst=[]
    for i in range(len(inp)):
        y_lst.append(w@inp[i]+b)
    return np.array(y_lst)
```

```
X_test=sc.fit_transform(x_test)
y_pred=predict(X_test)
```

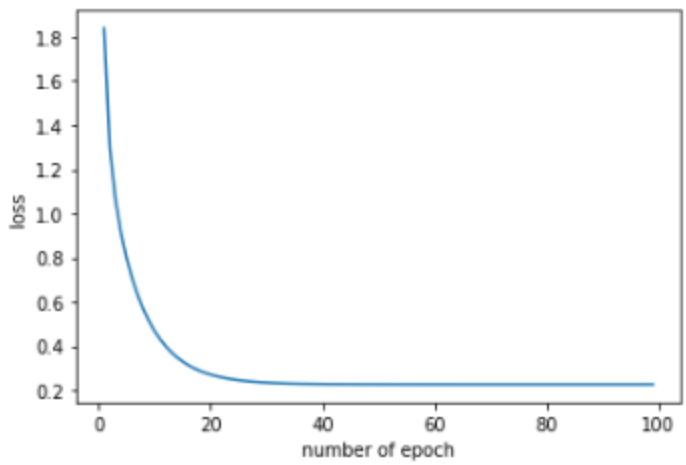
```
from sklearn.metrics import mean_squared_error, r2_score, mean_absolute_error
from math import sqrt
print("R2 score : %.3f" % r2_score(y_test, y_pred))
print("MSE: %.3f" % mean_squared_error(y_test, y_pred))
print("RMSE: %.3f" % sqrt(mean_squared_error(y_test, y_pred)))
print("MAE: %.3f" % sqrt(mean_absolute_error(y_test, y_pred)))
```

```
R2 score : 0.013
MSE: 0.227
RMSE: 0.476
MAE: 0.555
```

Fig: Gradient Plot for (1000,0.1) and calculation of errors

Here the gradient function (Loss or Cost) gradually decreases and after a point become constant and all kind of error values are calculated.

- (1000,0.001)




```
print("weight:",w)
print("bias:",b)
```

```
weight: [-0.27372976 -0.00738283  0.42391624  0.11013828 -0.10068015 -0.20637121
 -0.29480946]
bias: 3.6250277086579956
```

```
def predict(inp):
    y_lst=[]
    for i in range(len(inp)):
        y_lst.append(w@inp[i]+b)
    return np.array(y_lst)
```

```
X_test=sc.fit_transform(x_test)
y_pred=predict(X_test)
```

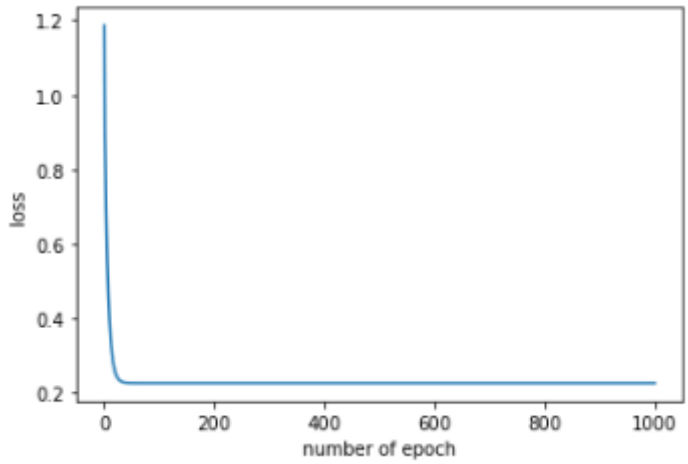
```
from sklearn.metrics import mean_squared_error, r2_score, mean_absolute_error
from math import sqrt
print("R2 score : %.3f" % r2_score(y_test, y_pred))
print("MSE: %.3f" % mean_squared_error(y_test, y_pred))
print("RMSE: %.3f" % sqrt(mean_squared_error(y_test, y_pred)))
print("MAE: %.3f" % sqrt(mean_absolute_error(y_test, y_pred)))
```

```
R2 score : -2.221
MSE: 0.740
RMSE: 0.860
MAE: 0.847
```

Fig: Gradient Plot for (1000,0.001) and calculation of errors

Here the gradient function (Loss or Cost) gradually decreases and after a point become constant and all kind of error values are calculated.

- (1000,0.5)




```
print("weight:",w)
print("bias:",b)
```

```
weight: [ 0.02027734  0.03124788  0.00584125  0.02827863 -0.01317883  0.00128798
 -0.01548743]
bias: 4.1910961022344475
```

```
def predict(inp):
    y_lst=[]
    for i in range(len(inp)):
        y_lst.append(w@inp[i]+b)
    return np.array(y_lst)
```

```
X_test=sc.fit_transform(x_test)
y_pred=predict(X_test)
```

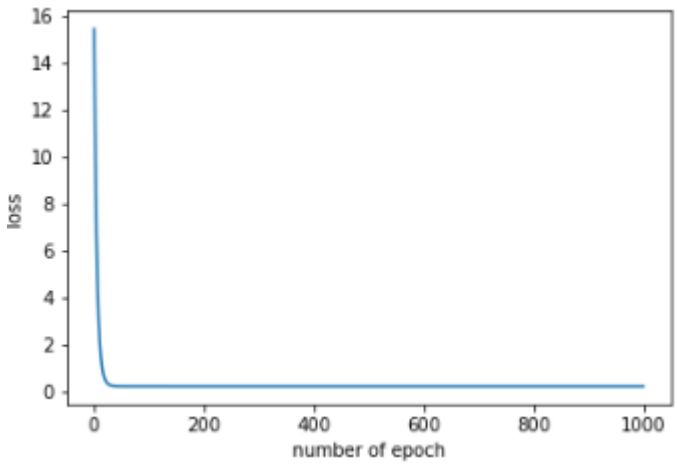
```
from sklearn.metrics import mean_squared_error, r2_score, mean_absolute_error
from math import sqrt
print("R2 score : %.3f" % r2_score(y_test, y_pred))
print("MSE: %.3f" % mean_squared_error(y_test, y_pred))
print("RMSE: %.3f" % sqrt(mean_squared_error(y_test, y_pred)))
print("MAE: %.3f" % sqrt(mean_absolute_error(y_test, y_pred)))
```

```
R2 score : 0.013
MSE: 0.227
RMSE: 0.476
MAE: 0.555
```

Fig: Gradient Plot for (1000,0.5) and calculation of errors

Here the gradient function (Loss or Cost) gradually decreases and after a point become constant and all kind of error values are calculated.

- (1000,0.05)



```
print("weight:",w)
print("bias:",b)
```

```
weight: [ 0.02027734  0.03124788  0.00584125  0.02827863 -0.01317883  0.00128798
 -0.01548743]
bias: 4.191096102234443
```

```
def predict(inp):
    y_lst=[]
    for i in range(len(inp)):
        y_lst.append(w@inp[i]+b)
    return np.array(y_lst)
```

```
X_test=sc.fit_transform(x_test)
y_pred=predict(X_test)
```

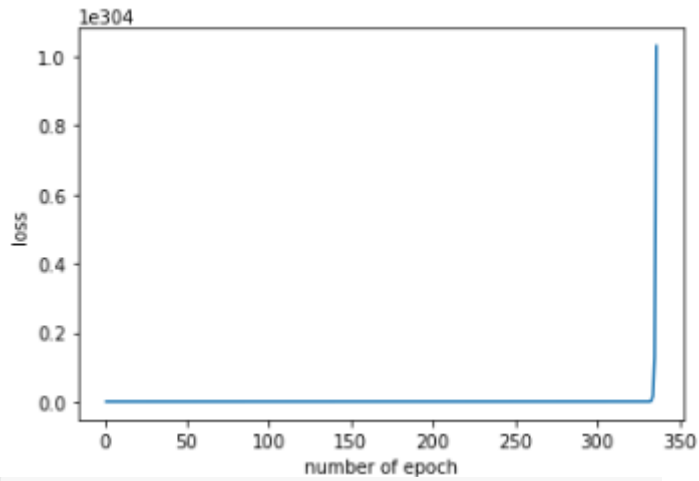
```
from sklearn.metrics import mean_squared_error, r2_score, mean_absolute_error
from math import sqrt
print("R2 score : %.3f" % r2_score(y_test, y_pred))
print("MSE: %.3f" % mean_squared_error(y_test, y_pred))
print("RMSE: %.3f" % sqrt(mean_squared_error(y_test, y_pred)))
print("MAE: %.3f" % sqrt(mean_absolute_error(y_test, y_pred)))
```

```
R2 score : 0.013
MSE: 0.227
RMSE: 0.476
MAE: 0.555
```

Fig: Gradient Plot for (1000,0.05) and calculation of errors

Here the gradient function (Loss or Cost) gradually decreases and after a point become constant and all kind of error values are calculated.

- (1000,1)




```
print("weight:",w)
print("bias:",b)
```

```
weight: [nan nan nan nan nan nan nan]
bias: nan
```

```
def predict(inp):
    y_lst=[]
    for i in range(len(inp)):
        y_lst.append(w@inp[i]+b)
    return np.array(y_lst)
```

```
X_test=sc.fit_transform(x_test)
y_pred=predict(X_test)
```

```
from sklearn.metrics import mean_squared_error, r2_score, mean_absolute_error
from math import sqrt
print("R2 score : %.3f" % r2_score(y_test, y_pred))
print("MSE: %.3f" % mean_squared_error(y_test, y_pred))
print("RMSE: %.3f" % sqrt(mean_squared_error(y_test, y_pred)))
print("MAE: %.3f" % sqrt(mean_absolute_error(y_test, y_pred)))
```

Fig: Gradient Plot for (1000,1) and calculation of errors

Here the gradient function (Loss or Cost) is constant in the beginning and after 320 iterations it starts increasing gradually and all kind of error values are calculated.

4.5. Regularization

4.5.1. Ridge Regression

- On Reviews

```
from sklearn.metrics import mean_squared_error, r2_sco
from math import sqrt
print(ridgeReg.coef_)
print("Intercept: %0.3f" % ridgeReg.intercept_)
print("R2 score : %.3f" % r2_score(y_test,ridgeReg.pre
print("MSE: %.3f" % mean_squared_error(y_test, ridgeRe
print("RMSE: %.3f" % sqrt(mean_squared_error(y_test, r
print("MAE: %.3f" % sqrt(mean_absolute_error(y_test, r
```

```
[0.03102091]
Intercept: 4.191
R2 score : 0.005
MSE: 0.226
RMSE: 0.475
MAE: 0.558
```

Fig: Error Calculation

OLS Regression Results

Dep. Variable:	Rating	R-s	
Model:	OLS	Adj. R-s	
Method:	Least Squares	F-s	
Date:	Sat, 05 Feb 2022	Prob (F-st	
Time:	05:42:26	Log-Lik	
No. Observations:	7587		
Df Residuals:	7585		
Df Model:	1		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[0.0
const	4.1908	0.006	761.478	0.000	4.1
Reviews	0.0326	0.005	5.997	0.000	0.0

Omnibus:	3172.793	Durbin-Watson:	
Prob(Omnibus):	0.000	Jarque-Bera (JB):	
Skew:	-1.920	Prob(JB):	
Kurtosis:	9.673	Cond. No.	

Fig: Summary using OLS for Reviews

Here we have applied Ridge Regression on Reviews with alpha value=0.05 and also calculated various types of errors as well as AIC, BIC etc. values with the help of ordinary least square and can be clearly visible from the summary figure of OLS.

- On Installs

```
from sklearn.metrics import mean_squared_error, r2_score
from math import sqrt
print(ridgeReg.coef_)
print("Intercept: %0.3f" % ridgeReg.intercept_)
print("R2 score : %.3f" % r2_score(y_test,ridgeReg.predi
print("MSE: %.3f" % mean_squared_error(y_test, ridgeReg.
print("RMSE: %.3f" % sqrt(mean_squared_error(y_test, ric
print("MAE: %.3f" % sqrt(mean_absolute_error(y_test, ric
```

```
[0.0227551]
Intercept: 4.191
R2 score : 0.003
MSE: 0.226
RMSE: 0.476
MAE: 0.559
```

Fig: Error Calculation

OLS Regression Results

Dep. Variable:	Rating	R-	
Model:	OLS	Adj. R-	
Method:	Least Squares	F-	
Date:	Sat, 05 Feb 2022	Prob (F-s	
Time:	05:42:36	Log-Lik	
No. Observations:	7587		
Df Residuals:	7585		
Df Model:	1		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[0.0
const	4.1906	0.006	760.635	0.000	4.1
Installs	0.0239	0.005	4.488	0.000	0.0

Omnibus:	3160.891	Durbin-Watson:	
Prob(Omnibus):	0.000	Jarque-Bera (JB):	
Skew:	-1.913	Prob(JB):	
Kurtosis:	9.640	Cond. No.	

Fig: Summary using OLS for Installs

Here we have applied Ridge Regression on Installs with alpha value=0.05 and also calculated various types of errors as well as AIC, BIC etc. values with the help of ordinary least square and can be clearly visible from the summary figure of OLS.

- On Size

```
from sklearn.metrics import mean_squared_error, r2_sc
from math import sqrt
print(ridgeReg.coef_)
print("Intercept: %0.3f" % ridgeReg.intercept_)
print("R2 score : %.3f" % r2_score(y_test,ridgeReg.pr
print("MSE: %.3f" % mean_squared_error(y_test, ridgeR
print("RMSE: %.3f" % sqrt(mean_squared_error(y_test,
print("MAE: %.3f" % sqrt(mean_absolute_error(y_test,
```

```
[0.03738379]
Intercept: 4.191
R2 score : 0.009
MSE: 0.225
RMSE: 0.474
MAE: 0.557
```

Fig: Error Calculation

OLS Regression Results

Dep. Variable:	Rating		
Model:	OLS	**Adj.**	
Method:	Least Squares		
Date:	Sat, 05 Feb 2022	**Prob (**	
Time:	05:42:45	**Log-**	
No. Observations:	7587		
Df Residuals:	7585		
Df Model:	1		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[0.0
const	4.1908	0.005	762.241	0.000	4.1
Size	0.0393	0.005	7.156	0.000	0.0

Omnibus:	3138.109	**Durbin-Watson**	
Prob(Omnibus):	0.000	**Jarque-Bera (JB)**	
Skew:	-1.897	**Prob(JB)**	
Kurtosis:	9.624	**Cond. No**	

Fig: Summary using OLS for Size

Here we have applied Ridge Regression on Size with alpha value=0.05 and also calculated various types of errors as well as AIC, BIC etc. values with the help of ordinary least square and can be clearly visible from the summary figure of OLS.

4.5.2. Lasso Regression

- On Reviews

```
from sklearn.metrics import mean_squared_error, r2_score
from math import sqrt
print("Coeffiecient: %0.3f" % lassoReg.coef_)
print("Intercept: %0.3f" % lassoReg.intercept_)
print("R2 score : %.3f" % r2_score(y_test,lassoReg.predi
print("MSE: %.3f" % mean_squared_error(y_test, lassoReg.
print("RMSE: %.3f" % sqrt(mean_squared_error(y_test, las
print("MAE: %.3f" % sqrt(mean_absolute_error(y_test, las
```

```
Coeffiecient: 0.000
Intercept: 4.191
R2 score : -0.000
MSE: 0.227
RMSE: 0.476
MAE: 0.560
```

Fig: Error Calculation

OLS Regression Results

Dep. Variable:	Rating	**R-s**
Model:	OLS	**Adj. R-s**
Method:	Least Squares	**F-s**
Date:	Sat, 05 Feb 2022	**Prob (F-s**
Time:	05:42:57	**Log-Lik**
No. Observations:	7587	
Df Residuals:	7585	
Df Model:	1	
Covariance Type:	nonrobust	

	coef	std err	t	P>|t|	[0.(
const	4.1908	0.006	761.478	0.000	4.1
Reviews	0.0326	0.005	5.997	0.000	0.(

Omnibus:	3172.793	**Durbin-Watson:**	
Prob(Omnibus):	0.000	**Jarque-Bera (JB):**	
Skew:	-1.920	**Prob(JB):**	
Kurtosis:	9.673	**Cond. No.**	

Fig: Summary using OLS for Reviews

Here we have applied Lasso Regression on Reviews with alpha value=0.05 and also calculated various types of errors as well as AIC, BIC etc. values with the help of ordinary least square and can be clearly visible from the summary figure of OLS.

- On Installs

```
from sklearn.metrics import mean_squared_error, r2_score
from math import sqrt
print("Coeffiecient: %0.3f" % lassoReg.coef_)
print("Intercept: %0.3f" % lassoReg.intercept_)
print("R2 score : %.3f" % r2_score(y_test,lassoReg.predi
print("MSE: %.3f" % mean_squared_error(y_test, lassoReg.
print("RMSE: %.3f" % sqrt(mean_squared_error(y_test, las
print("MAE: %.3f" % sqrt(mean_absolute_error(y_test, las
```

```
Coeffiecient: 0.000
Intercept: 4.191
R2 score : -0.000
MSE: 0.227
RMSE: 0.476
MAE: 0.560
```

Fig: Error Calculation

OLS Regression Results

Dep. Variable:	Rating	**R-s**
Model:	OLS	**Adj. R-s**
Method:	Least Squares	**F-s**
Date:	Sat, 05 Feb 2022	**Prob (F-st**
Time:	05:43:06	**Log-Like**
No. Observations:	7587	
Df Residuals:	7585	
Df Model:	1	
Covariance Type:	nonrobust	

	coef	std err	t	P>|t|	[0.02
const	4.1906	0.006	760.635	0.000	4.18
Installs	0.0239	0.005	4.488	0.000	0.01

Omnibus:	3160.891	**Durbin-Watson:**	
Prob(Omnibus):	0.000	**Jarque-Bera (JB):**	
Skew:	-1.913	**Prob(JB):**	
Kurtosis:	9.640	**Cond. No.**	

Fig: Summary using OLS for Installs

Here we have applied Lasso Regression on Installs with alpha value=0.05 and also calculated various types of errors as well as AIC, BIC etc. values with the help of ordinary least square and can be clearly visible from the summary figure of OLS.

- On Size

```
from sklearn.metrics import mean_squared_error, r2_score,
from math import sqrt
print("Coeffiecient: %0.3f" % lassoReg.coef_)
print("Intercept: %0.3f" % lassoReg.intercept_)
print("R2 score : %.3f" % r2_score(y_test,lassoReg.predict
print("MSE: %.3f" % mean_squared_error(y_test, lassoReg.pr
print("RMSE: %.3f" % sqrt(mean_squared_error(y_test, lasso
print("MAE: %.3f" % sqrt(mean_absolute_error(y_test, lasso
```

```
Coeffiecient: 0.000
Intercept: 4.191
R2 score : -0.000
MSE: 0.227
RMSE: 0.476
MAE: 0.560
```

Fig: Error Calculation

OLS Regression Results

Dep. Variable:	Rating	**R-sq**	
Model:	OLS	**Adj. R-sq**	
Method:	Least Squares	**F-sta**	
Date:	Sat, 05 Feb 2022	**Prob (F-sta**	
Time:	05:43:14	**Log-Likel**	
No. Observations:	7587		
Df Residuals:	7585		
Df Model:	1		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[0.025
const	4.1908	0.005	762.241	0.000	4.180
Size	0.0393	0.005	7.156	0.000	0.029

Omnibus:	3138.109	**Durbin-Watson:**	
Prob(Omnibus):	0.000	**Jarque-Bera (JB):**	1
Skew:	-1.897	**Prob(JB):**	
Kurtosis:	9.624	**Cond. No.**	

Fig: Summary using OLS for Size

Here we have applied Lasso Regression on Size with alpha value=0.05 and also calculated various types of errors as well as AIC, BIC etc. values with the help of ordinary least square and can be clearly visible from the summary figure of OLS.

5. CONCLUSION

In this report, we discussed the importance of all the above stated regression algorithms and the challenges to analyzing preprocessed data and predicting the dependent variables with the help of machine learning and deep learning algorithms. We have also given the solution of this challenge in the form of various regression techniques which are suitable according to the dataset in which we want to do work. We explained and demonstrated all the regression algorithms, with the help of which we have easily visualized the importance of each technique. Our dataset also helps us to measure predicted values of the dependent variables according to their correlation with the independent variables.

After cleaning the original data and applying various kind of regression techniques, we came to conclusion that Random Forest Regressor is the best regression technique for this model among all the techniques we have applied. We have also concluded that Gradient Descent values does not depend on type of regression technique. In Polynomial Regression, there are some changes on changing degree of the regression because as the regression degree increases, it covers more

actual value. We have noted the AIC, BIC, CP and adjusted R square value for each condition after which we have also applied Ridge and Lasso Regression on the 3 strongly correlated feature from the data. Thus, we have completely analyzed each and every step, noted down each type of error value with the help of which we can predict 'Rating' of any app accurately.

6. REFERENCES

- https://www.kaggle.com/lava18/google-play-store-apps
- https://medium.com/analytics-vidhya/implementing-gradient-descent-for-multi-linear-regression-from-scratch-3e31c114ae12
- https://www.analyticsvidhya.com/blog/2021/10/understanding-polynomial-regression-model/
- https://data36.com/polynomial-regression-python-scikit-learn/
- https://www.investopedia.com/ask/answers/060315/what-difference-between-linear-regression-and-multiple-regression.asp#:~:text=Linear%20regression%20attempts%20to%20draw,called%20a%20multiple%20linear%20regression.
- https://towardsdatascience.com/ridge-and-lasso-regression-a-complete-guide-with-python-scikit-learn-e20e34bcbf0b

TABLE (WHICH WAS REQUIRED IN THE END OF THE PROJECT REPORT) FILLED UP WITH DETAILS OF VALUES OF ALL THE TASKS WHICH WE HAVE PERFORMED HAS BEEN SUBMITTED SEPARATELY IN DOCS FORM AS IT WAS INTERFERRING WITH THE ORIENTATION OF THE PROJECT REPORT.