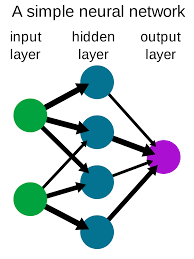
Neural Network Introduction

A neural network is a series of algorithms that endeavors to recognize basic relationships in a set of record through a process that mimics the way the human brain operates. In this method, neural networks defines systems of neurons, either organic or artificial.

Neural Networks are analytic techniques modeled after the (hypothesized) processes of learning in the cognitive system and the neurological functions of the brain and capable of predicting new observations (on specific variables) from other observations after implementing a process of so-called learning from existing information. Neural Networks are one of the Data Mining techniques.

The first phase is to design a specific network architecture (that involves a definite number of “layers” each including a specific number of “neurons”). The size and structure of the network need to match the nature (e.g., the formal complexity) of the investigated phenomenon. Because the latter is not known very well at this early stage, this task is not easy and often involves multiple “trials and errors.”



The new network is then subjected to the process of “training.” In that phase, neurons apply an iterative process to the number of inputs (variables) to adjust the weights of the network to optimally predict (in traditional terms one could say, find a “fit” to) the sample data on which the “training” is performed. After the phase of learning from an existing data set, the new network is ready and it can then be used to generate predictions.

Neural networks have view an eruption of interest over the last few years, and are being successfully used across an extraordinary area of problem domains, in areas as diverse as finance, medicine, engineering, geology, and physics. There are two elements of a neural network which are as follows −

**Power** − Neural networks are very refined modeling techniques adequate of modeling extremely complex functions. In particular, neural networks are nonlinear. For some years linear modeling has been the generally used methods in most modeling domains because linear models have well-known optimization strategies.

**Ease of use** − Neural networks learn by example. The neural network user gathers representative data and then invokes training algorithms to automatically learn the structure of the data.

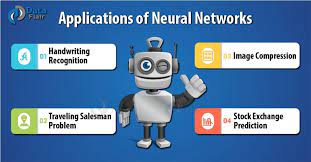
Although the user does required to have some heuristic knowledge of how to choose and prepare records, how to choose an appropriate neural network, and how to execute the results, the level of user knowledge needed to successfully use neural networks is much lower than would be the case using (for instance) some more traditional nonlinear statistical methods.

There are various applications of Neural Networks which are as follows −

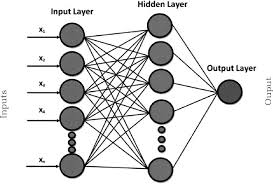
* **Detection of medical phenomena** − An array of health-related indices (e.g., a combination of heart rate, levels of several substances in the blood, respiration rate) can be checked. The onset of a specific medical condition can be related to a very complex (e.g., nonlinear and interactive) combination of changes on a subset of the variables being checked. Neural networks have been used to identify this predictive design so that the appropriate treatment can be recommended.
* **Stock market prediction** − Fluctuations of stock prices and stock indices are another example of a complex, multidimensional, but in some circumstances at least partially deterministic phenomenon. Neural networks are being used by many technical analysts to make predictions about stock prices based upon a large number of factors such as the past performance of other stocks and various economic indicators.
* **Credit assignment** − A variety of pieces of data are generally known about an applicant for a loan. For example, the applicant’s age, education, occupation, and some other facts can be accessible. After training a neural network on historical data, neural network analysis can identify the most relevant characteristics and use those to classify applicants as good or bad credit risks.
* **Monitoring the condition of machinery** − Neural networks can be instrumental in cutting values by carrying additional expertise to scheduling the preventive preservation of machines.

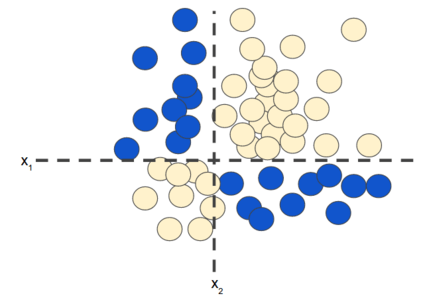
A neural network can be trained to categorize between the sounds a machine creates when it is running generally (“false alarms”) versus when it is on the verge of a issue. After this training period, the proficiency of the network can be used to address a technician of an upcoming breakdown, before it appears and generates costly unforeseen “downtime.”

* **Engine management** − Neural networks have been used to evaluate the input of sensors from an engine. The neural network controls the various parameters within which the engine functions, to achieve a particular goal, such as minimizing fuel consumption.

****

**Neural Networks: Structure**



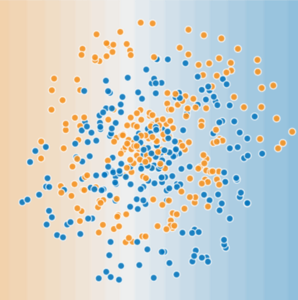


**Figure 1. Nonlinear classification problem.**

"Nonlinear" means that you can't accurately predict a label with a model of the form

In other words, the "decision surface" is not a line. Previously, we looked at [feature crosses](https://developers.google.com/machine-learning/crash-course/feature-crosses/video-lecture) as one possible approach to modeling nonlinear problems.

Now consider the following data set:



**Figure 2. A more difficult nonlinear classification problem.**

The data set shown in Figure 2 can't be solved with a linear model.

To see how neural networks might help with nonlinear problems, let's start by representing a linear model as a graph:

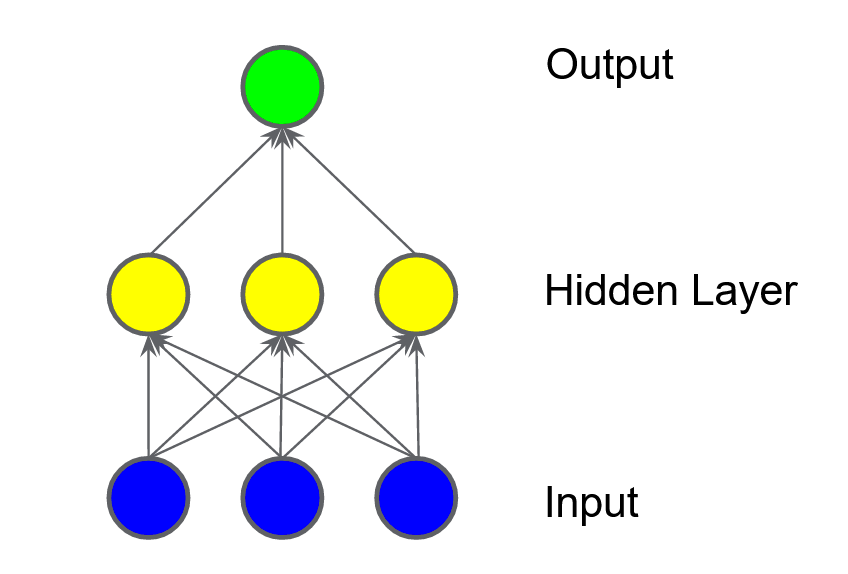
**Figure 3. Linear model as graph.**

Each blue circle represents an input feature, and the green circle represents the weighted sum of the inputs.

How can we alter this model to improve its ability to deal with nonlinear problems?

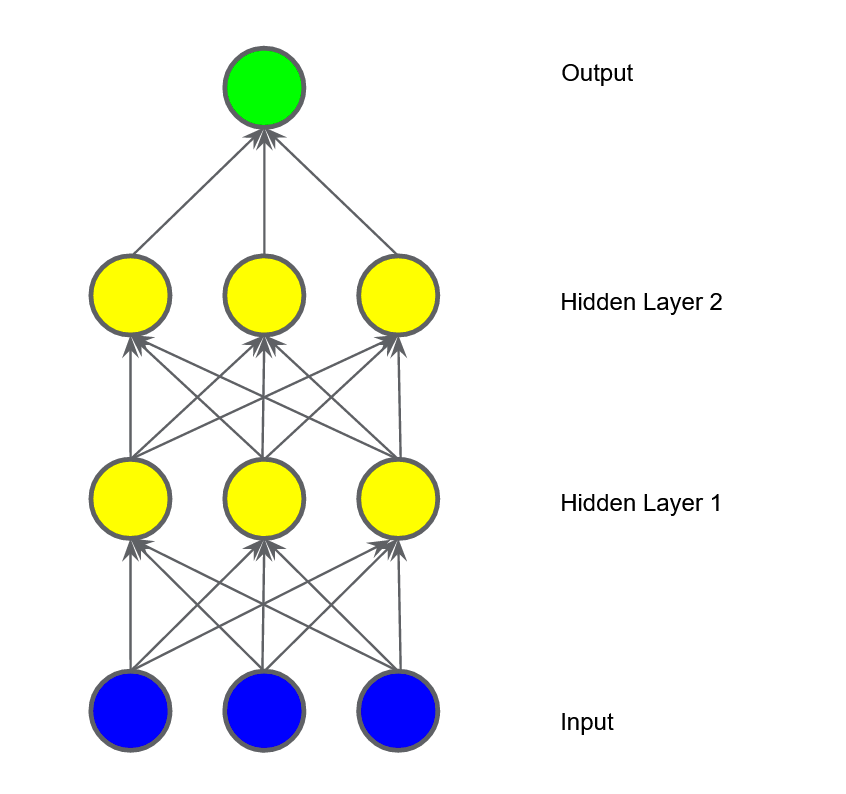
**Hidden Layers**

In the model represented by the following graph, we've added a "hidden layer" of intermediary values. Each yellow node in the hidden layer is a weighted sum of the blue input node values. The output is a weighted sum of the yellow nodes.



Is this model linear? Yes—its output is still a linear combination of its inputs.

In the model represented by the following graph, we've added a second hidden layer of weighted sums.



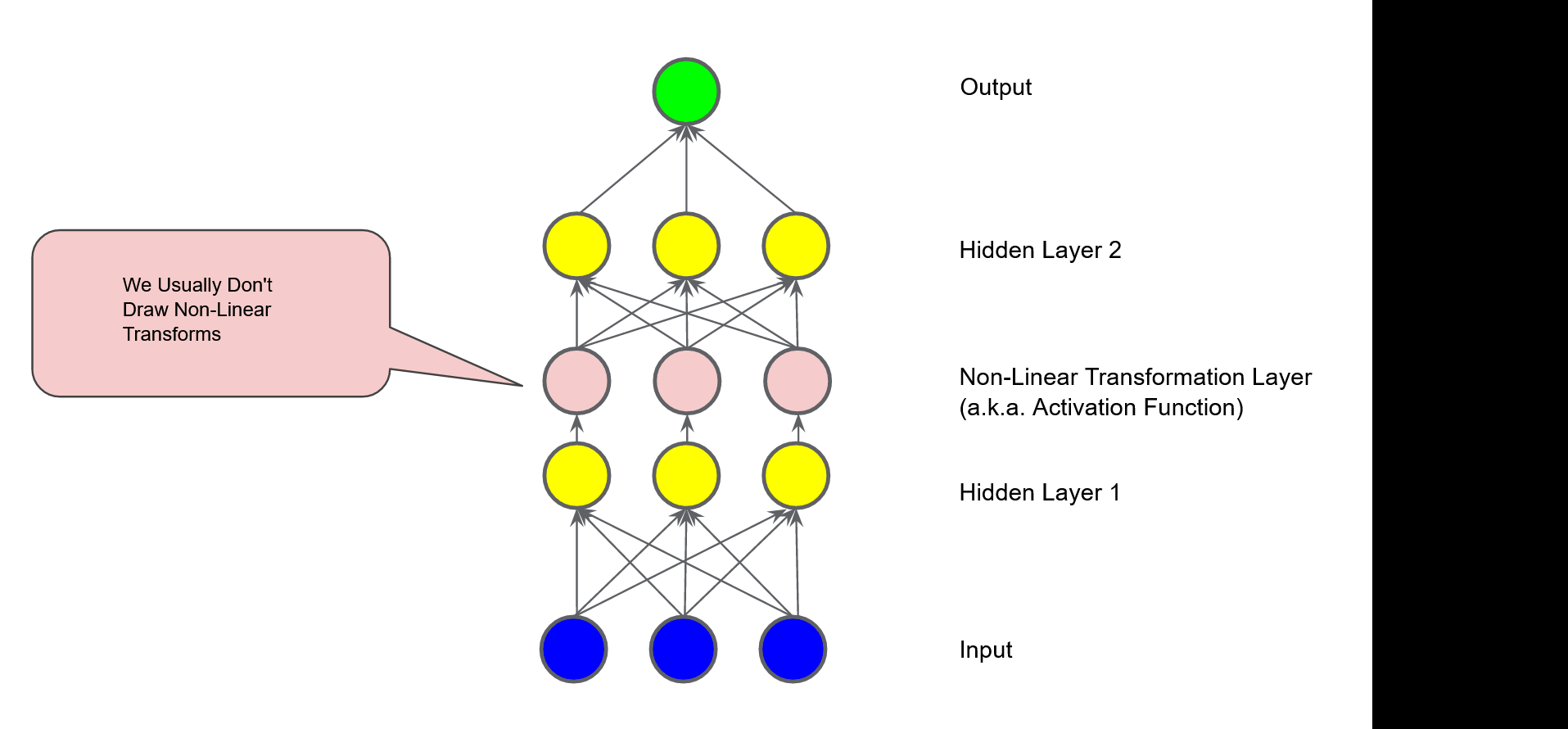
**Figure 5. Graph of three-layer model.**

Is this model still linear? Yes, it is. When you express the output as a function of the input and simplify, you get just another weighted sum of the inputs. This sum won't effectively model the nonlinear problem in Figure 2.

**Activation Functions**

To model a nonlinear problem, we can directly introduce a nonlinearity. We can pipe each hidden layer node through a nonlinear function.

In the model represented by the following graph, the value of each node in Hidden Layer 1 is transformed by a nonlinear function before being passed on to the weighted sums of the next layer. This nonlinear function is called the activation function.



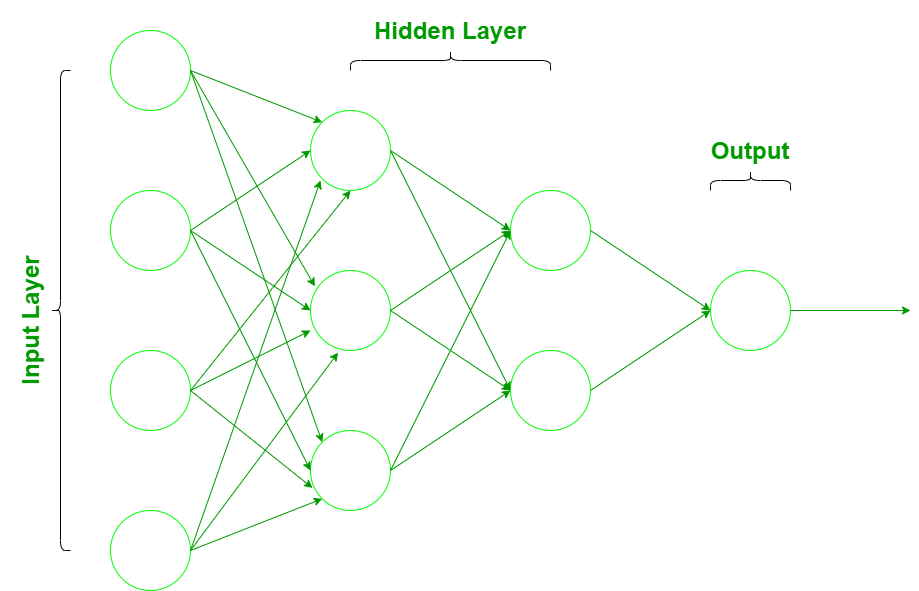
**Figure 6. Graph of three-layer model with activation function.**

Now that we've added an activation function, adding layers has more impact. Stacking nonlinearities on nonlinearities lets us model very complicated relationships between the inputs and the predicted outputs. In brief, each layer is effectively learning a more complex, higher-level function over the raw inputs. If you'd like to develop more intuition on how this works, see [Chris Olah's excellent blog post](http://colah.github.io/posts/2014-03-NN-Manifolds-Topology/).

#### The Architecture of Neural Networks

A neural network consists of three layers:

1. **Input Layer:** Layers that take inputs based on existing data.
2. **Hidden Layer:** Layers that use backpropagation to optimise the weights of the input variables in order to improve the predictive power of the model.
3. **Output Layer:** Output of predictions based on the data from the input and hidden layers.



## How to train a neural network in R?

### Step 1: Define the training set.

#creating training data set  
Sci = c(70,71,72,73,68,69,65,69,80,68)  
Mat = c(91,92,93,94,65,69,61,55,91,79)  
Eng = c(82,83,84,85,73,66,50,62,95,68)  
Pass = c(1,1,1,1,0,0,0,0,1,0)  
df=data.frame(Sci,Mat,Eng,Pass)

We are creating our sample dataframe with three subjects- Science, Maths, English, and a Pass column which indicates if the student has passed or not. One shows pass, and 0 shows fail. The model should be able to predict if the student passes or fails.

### Step 2: Install and load neuralnet package

#installing and loading neuralnet package  
install.packages("neuralnet")  
library("neuralnet")

### Step 3: Fit neural network

#fitting the model  
nn=neuralnet(Pass~Sci+Mat+Eng,data=df, hidden=3,act.fct = "logistic",  
linear.output = FALSE)  
#Pass~Sci+Mat+Eng, Pass is label and Sci,Mat,Eng are features.  
#df is dataframe,  
#hidden=3 stands for a single hidden layer with 3 neurons   
#act.fct = "logistic" is used for smoothing the result.  
#linear.ouput=FALSE is set FALSE for apply act.fct

### Step 4: Plot the neural network

plot(nn)

### Step 5: Create a test dataset

#creating tesing set  
sci = c(80,75,65,68)  
mat = c(95,92,69,45)  
eng = c(85,83,55,50)  
test=data.frame(sci,mat,eng)

### Step 6: Predict results for the test dataset

#prediction  
predict=compute(nn,test)  
predict$net.result  
  
probab<- predict$net.result  
  
#converting probabilities into 1 and 0  
pre <- ifelse(probab>0.5, 1, 0)  
pre

[,1]

[1,] 1

[2,] 1

[3,] 1

[4,] 0

The predicted output for the testing set is 1, 1, 1, 0.   
The neural network can be used for similarly making predictions for real datasets.

The Neural Network is constructed from 3 type of layers:

1. Input layer — initial data for the neural network.
2. Hidden layers — intermediate layer between input and output layer and place where all the computation is done.
3. Output layer — produce the result for given inputs.

**Back propagation**

Backpropagation is a process involved in training a neural network. It involves taking the error rate of a forward propagation and feeding this loss backward through the neural network layers to fine-tune the weights.

Backpropagation is the essence of neural net training. It is the practice of fine-tuning the weights of a neural net based on the error rate (i.e. loss) obtained in the previous epoch (i.e. iteration.) Proper tuning of the weights ensures lower error rates, making the model reliable by increasing its generalization.

## How to Set the Model Components for a Backpropagation Neural Network

Imagine that we have a deep neural network that we need to train. The purpose of training is to build a model that performs the exclusive OR (XOR) functionality with two inputs and three hidden units, such that the training set (truth table) looks something like the following:

X1 | X2 | Y

0 | 0 | 0

0 | 1 | 1

1 | 0 | 1

1 | 1 | 0

We also need an [activation function](https://builtin.com/machine-learning/activation-functions-deep-learning) that determines the activation value at every node in the neural net. For simplicity, let’s choose an identity [activation function](https://builtin.com/machine-learning/relu-activation-function):f(a) = a

We also need a hypothesis function that determines the input to the activation function. This function is going to be the ever-famous:

h(X) = W0.X0 + W1.X1 + W2.X2

            or

h(X) = sigma(W.X) for all (W, X)

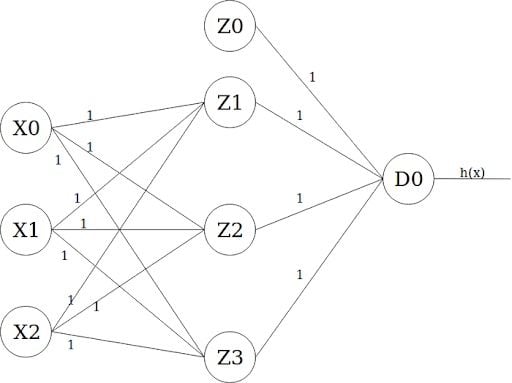
Let’s also make the loss function the usual cost function of logistic regression. It looks a bit complicated, but it’s actually fairly simple:

cost function of linear logistic regression equationCost function of logistic regression equation. | Image: Anas Al-Masri

We’re going to use the batch [gradient descent optimization function](https://builtin.com/data-science/gradient-descent) to determine in what direction we should adjust the weights to get a lower loss than our current one. Finally, we’ll set the learning rate to 0.1 and all the weights will be initialized to one.

## Building a Neural Network

Let’s finally draw a diagram of our long-awaited neural net. It should look something like this:

Model of a neural network. | Image: Anas Al-Masri

The leftmost layer is the [input layer](https://builtin.com/machine-learning/fully-connected-layer), which takes X0 as the bias term of value one, and X1 and X2 as input features. The layer in the middle is the first hidden layer, which also takes a bias term Z0 value of one. Finally, the output layer has only one output unit D0 whose activation value is the actual output of the model (i.e. h(x).)

## How Forward Propagation Works

It is now the time to feed-forward the information from one layer to the next. This goes through two steps that happen at every node/unit in the network:

1. Getting the weighted sum of inputs of a particular unit using the h(x) function we defined earlier.
2. Plugging the value we get from step one into the activation function, we have (f(a)=a, in this example) and using the activation value we get the output of the activation function as the input feature for the connected nodes in the next layer.

Units X0, X1, X2 and Z0 do not have any units connected to them providing inputs. Therefore, the steps mentioned above do not occur in those nodes. However, for the rest of the nodes/units, this is how it all happens throughout the neural net for the first input sample in the training set:

Unit Z1:

h(x) = W0.X0 + W1.X1 + W2.X2

= 1 . 1 + 1 . 0 + 1 . 0

= 1 = a

z = f(a) = a => z = f(1) = 1

Same goes for the remaining units:

Unit Z2:

h(x) = W0.X0 + W1.X1 + W2.X2

= 1 . 1 + 1 . 0 + 1 . 0

= 1 = a

z = f(a) = a => z = f(1) = 1

Unit Z3:

h(x) = W0.X0 + W1.X1 + W2.X2

= 1 . 1 + 1 . 0 + 1 . 0

= 1 = a

z = f(a) = a => z = f(1) = 1

Unit D0:

h(x) = W0.Z0 + W1.Z1 + W2.Z2 + W3.Z3

= 1 . 1 + 1 . 1 + 1 . 1 + 1 . 1

= 4 = a

z = f(a) = a => z = f(4) = 4

As we mentioned earlier, the activation value (z) of the final unit (D0) is that of the whole model. Therefore, our model predicted an output of one for the set of inputs {0, 0}. Calculating the loss/cost of the current iteration would follow:

Loss = actual\_y - predicted\_y

    =    0     -     4

    =    -4

The actual\_y value comes from the training set, while the predicted\_y value is what our model yielded. So the cost at this iteration is equal to -4.

## When Do You Use Backpropagation in Neural Networks?

According to our example, we now have a model that does not give [accurate predictions](https://builtin.com/machine-learning/bias-machine-learning). It gave us the value four instead of one and that is attributed to the fact that its weights have not been tuned yet. They’re all equal to one. We also have the loss, which is equal to -4. Backpropagation is all about feeding this loss backward in such a way that we can fine-tune the weights based on this. The optimization function, gradient descent in our example, will help us find the weights that will hopefully yield a smaller loss in the next iteration. So, let’s get to it.

If feeding forward happened using the following functions: f(a) = a

Optimization function in gradient descent equationOptimization function equation in a gradient descent. | Image: Anas Al-Masri

Then feeding backward will happen through the partial derivatives of those functions. There is no need to go through the equation to arrive at these derivatives. All we need to know is that the above functions will follow:

f'(a) = 1

J'(w) = Z . delta

Z is just the z value we obtained from the activation function calculations in the feed-forward step, while delta is the loss of the unit in the layer.

I know it’s a lot of information to absorb in one sitting, but I suggest you take your time to really understand what is going on at each step before going further.

[**Polynomial Regression**](https://www.geeksforgeeks.org/python-implementation-of-polynomial-regression/) is a form of linear regression in which the relationship between the independent variable x and dependent variable y is modeled as an nth degree polynomial. Polynomial regression fits a nonlinear relationship between the value of x and the corresponding conditional mean of y, denoted E(y|x). Basically it adds the quadratic or polynomial terms to the regression. Generally, this kind of regression is used for one resultant variable and one predictor.

#### Need of Polynomial Regression

* Unlike linear data set, if one tries to apply linear model on non-linear data set without any modification, then there will be a very unsatisfactory and drastic result .
* This may lead to increase in loss function, decrease in accuracy and high error rate.
* Unlike linear model, polynomial model covers more data points.

#### Applications of Polynomial Regression

Generally, polynomial regression is used in the following scenarios :

* Rate of growth of tissues.
* Progression of the epidemics related to disease.
* Distribution phenomenon of the isotopes of carbon in lake sediments.

#### Explanation of Polynomial Regression in R Programming

Polynomial Regression is also known as Polynomial Linear Regression since it depends on the linearly arranged coefficients rather than the variables. In [R](https://www.geeksforgeeks.org/introduction-to-r-programming-language/), if one wants to implement polynomial regression then he must install the following packages:

* **tidyverse** package for better visualization and manipulation.
* **caret** package for a smoother and easier machine learning workflow.

After proper installation of the packages, one needs to set the data properly. For that, first one needs to split the data into two sets(train set and test set). Then one can visualize the data into various plots. In R, in order to fit a polynomial regression, first one needs to generate pseudo random numbers using the [**set.seed(n)**](https://www.geeksforgeeks.org/generate-data-sets-of-same-random-values-in-r-programming-set-seed-function/) function.

The polynomial regression adds polynomial or quadratic terms to the regression equation as follow:

**medv = b0 + b1 \* lstat + b2 \* lstat 2**

**where**

**mdev:** is the median house value

**lstat:** is the predictor variable

In R, to create a predictor x2 one should use the function **I()**, as follow: **I(x2)**. This raise x to the power 2. The polynomial regression can be computed in R as follow:

lm(medv ~ lstat + I(lstat^2), data = train.data)

For this following example let’s take the Boston data set of MASS package. 

**Example:**

## r

|  |
| --- |
| # R program to illustrate  # Polynomial regression    # Importing required library  library(tidyverse)  library(caret)  theme\_set(theme\_classic())    # Load the data  data("Boston", package = "MASS")  # Split the data into training and test set  set.seed(123)  training.samples <- Boston$medv %>%    createDataPartition(p = 0.8, list = FALSE)  train.data  <- Boston[training.samples, ]  test.data <- Boston[-training.samples, ]    # Build the model  model <- lm(medv ~ poly(lstat, 5, raw = TRUE),              data = train.data)  # Make predictions  predictions <- model %>% predict(test.data)  # Model performance  modelPerfomance = data.frame(                      RMSE = RMSE(predictions, test.data$medv),                       R2 = R2(predictions, test.data$medv)                   )    print(lm(medv ~ lstat + I(lstat^2), data = train.data))  print(modelPerfomance) |

**Output:**

Call:

lm(formula = medv ~ lstat + I(lstat^2), data = train.data)

Coefficients:

(Intercept) lstat I(lstat^2)

42.5736 -2.2673 0.0412

RMSE R2

1 5.270374 0.6829474

#### **Graph plotting of Polynomial Regression**

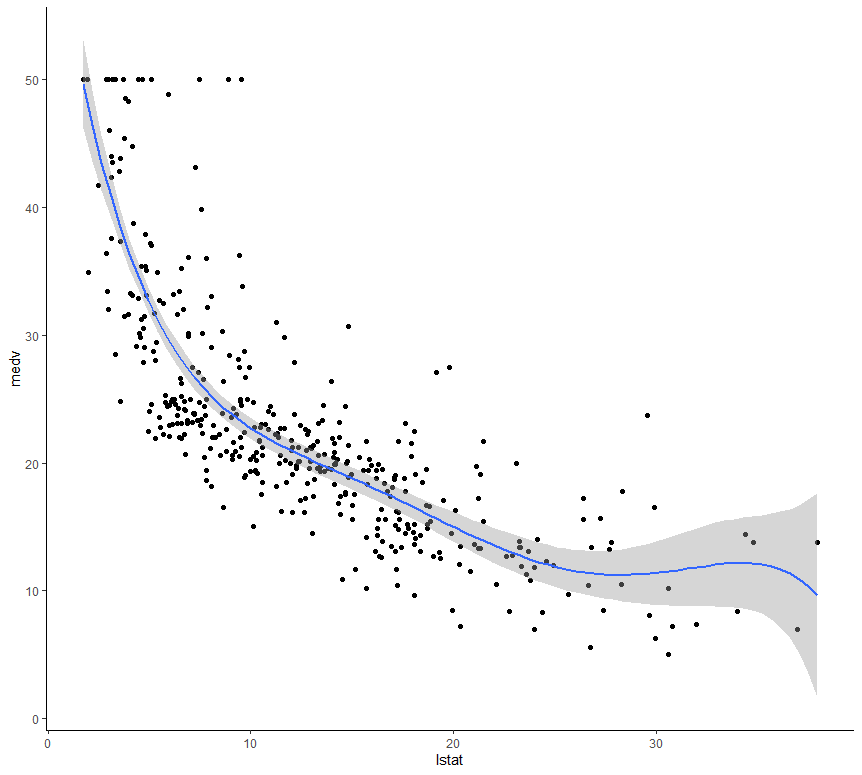
In R, if one wants to plot a graph for the output generated on implementing Polynomial Regression he can use the **ggplot()** function.

**Example:**

## r

|  |
| --- |
| # R program to illustrate  # Graph plotting in  # Polynomial regression    # Importing required library  library(tidyverse)  library(caret)  theme\_set(theme\_classic())    # Load the data  data("Boston", package = "MASS")  # Split the data into training and test set  set.seed(123)  training.samples <- Boston$medv %>%    createDataPartition(p = 0.8, list = FALSE)  train.data  <- Boston[training.samples, ]  test.data <- Boston[-training.samples, ]    # Build the model  model <- lm(medv ~ poly(lstat, 5, raw = TRUE), data = train.data)  # Make predictions  predictions <- model %>% predict(test.data)  # Model performance  data.frame(RMSE = RMSE(predictions, test.data$medv),             R2 = R2(predictions, test.data$medv))    ggplot(train.data, aes(lstat, medv) ) + geom\_point() +  stat\_smooth(method = lm, formula = y ~ poly(x, 5, raw = TRUE)) |

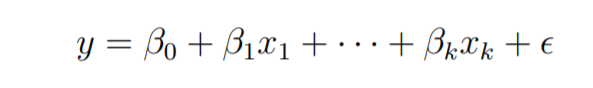
**Output:**



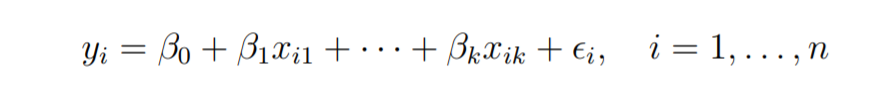
**Multiple Linear Regression :**   
It is the most common form of Linear Regression. Multiple Linear Regression basically describes how a single response variable Y depends linearly on a number of predictor variables.  
The basic examples where Multiple Regression can be used are as follows: 

1. The selling price of a house can depend on the desirability of the location, the number of bedrooms, the number of bathrooms, the year the house was built, the square footage of the lot, and a number of other factors.
2. The height of a child can depend on the height of the mother, the height of the father, nutrition, and environmental factors.

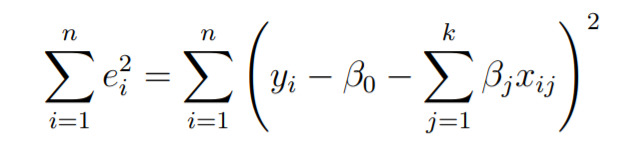
**Estimation of the Model Parameters**   
Consider a multiple linear Regression model with k independent predictor variable x1, x2……, xk, and one response variable y. 



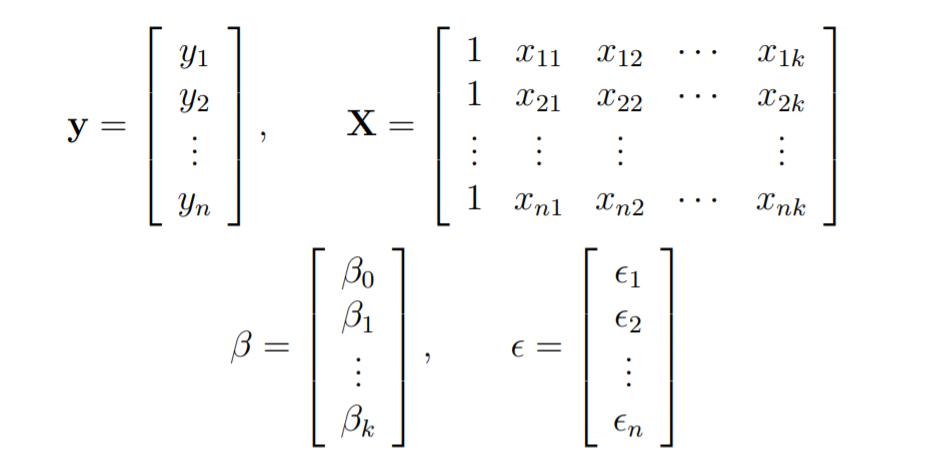
Suppose we have n observation on the k+1 variables and the variable of n should be greater than k. 



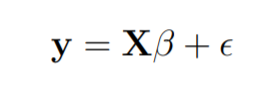
The basic goal in least-squares regression is to fit a hyper-plane into (k + 1)-dimensional space that minimizes the sum of squared residuals. 



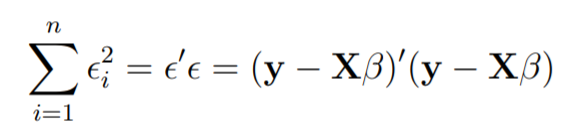
Before taking the derivative with respect to the model parameters set them equal to zero and derive the least-squares normal equations that the parameters would have to fulfill.   
These equations are formulated with the help of vectors and matrices.   
Let 



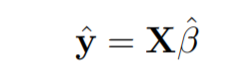
The linear Regression model is written in the form as follows: 



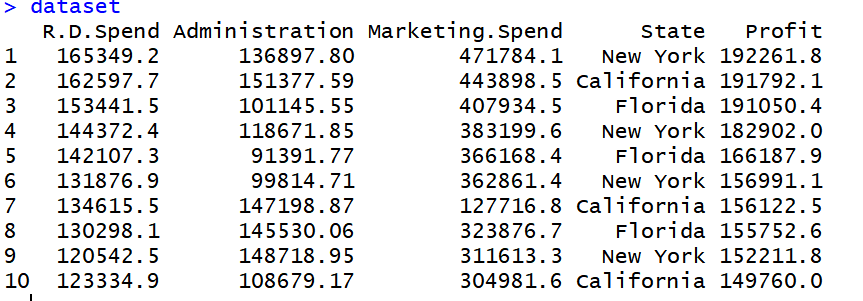
In linear regression the least square parameters estimate b 



Imagine the columns of X to be fixed, they are the data for a specific problem and say b to be variable. We want to find the “best” b in the sense that the sum of squared residuals is minimized.   
The smallest that the sum of squares could be is zero. 

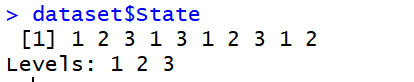


Here y is the estimated response vector.  
Following R code is used to implement Multiple Linear Regression on following dataset [data2](https://media.geeksforgeeks.org/wp-content/uploads/data2.csv).  
the dataset looks like this: 



## R

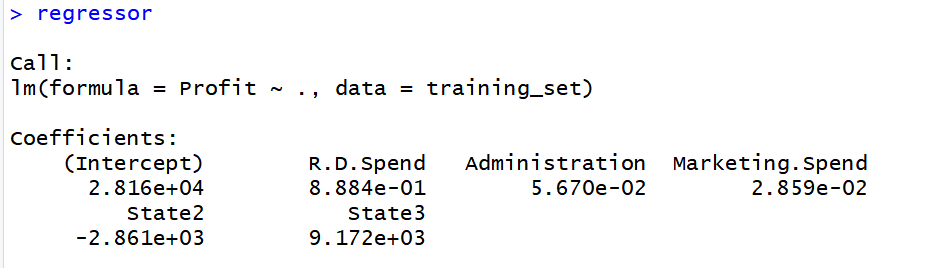
|  |
| --- |
| # Multiple Linear Regression    # Importing the dataset  dataset = read.csv('data2.csv')    # Encoding categorical data  dataset$State = factor(dataset$State,                         levels = c('New York', 'California', 'Florida'),                         labels = c(1, 2, 3))  dataset$State |

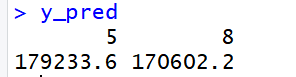


## R

|  |
| --- |
| # Splitting the dataset into the Training set and Test set  # install.packages('caTools')  library(caTools)  set.seed(123)  split = sample.split(dataset$Profit, SplitRatio = 0.8)  training\_set = subset(dataset, split == TRUE)  test\_set = subset(dataset, split == FALSE)    # Feature Scaling  # training\_set = scale(training\_set)  # test\_set = scale(test\_set)    # Fitting Multiple Linear Regression to the Training set  regressor = lm(formula = Profit ~ .,                 data = training\_set)    # Predicting the Test set results  y\_pred = predict(regressor, newdata = test\_set) |

Output: 





A Poisson Regression model is used to model count data and model response variables (Y-values) that are counts. It shows which X-values work on the Y-value and more categorically, it counts data: discrete data with non-negative integer values that count something.

In other words, it shows which explanatory variables have a notable effect on the response variable. Poisson Regression involves regression models in which the response variable is in the form of counts and not fractional numbers.

**Mathematical Equation:**

log(y) = a + b1x1 + b2x2 + bnxn.....

**Parameters:**

* **y:** This parameter sets as a response variable.
* **a** and **b:** The parameter a and b are the numeric coefficients.
* **x:** This parameter is the predictor variable.

#### Creating Poisson Regression Model

The function used to create the Poisson regression model is the **glm()** function.

**Syntax:** glm(formula, data, family)

**Parameters:**

* **formula:** This parameter is the symbol presenting the relationship between the variables.
* **data:** The parameter is the data set giving the values of these variables.
* **family:** This parameter R object to specify the details of the model. It’s value is ‘Poisson’ for Logistic Regression.

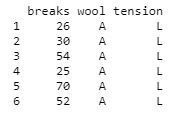
**Example:**

**Approach:** To understand how we can create:

* + We use the data set “warpbreaks”.
  + Considering “breaks” as the response variable.
  + The wool “type” and “tension” are taken as predictor variables.

**Code:**

|  |
| --- |
| input <- warpbreaks  print(head(input)) |

**Output:**  


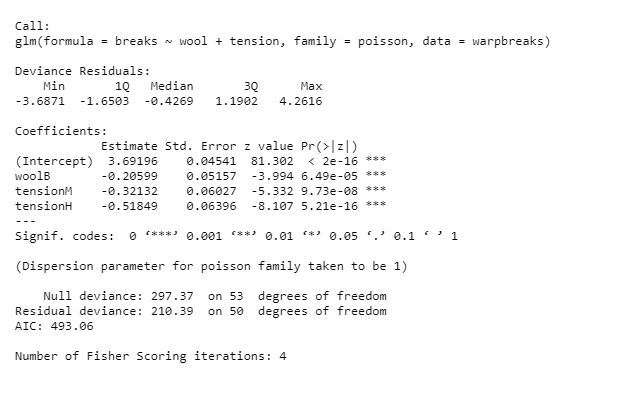
#### Create Regression Model

**Approach:** Creating the poisson regression model:

* + Take the parameters which are required to make model.
  + let’s use summary() function to find the summary of the model for data analysis.

**Example:**

|  |
| --- |
| output <-glm(formula = breaks ~ wool + tension,               data = warpbreaks, family = poisson)  print(summary(output)) |

**Output:**  


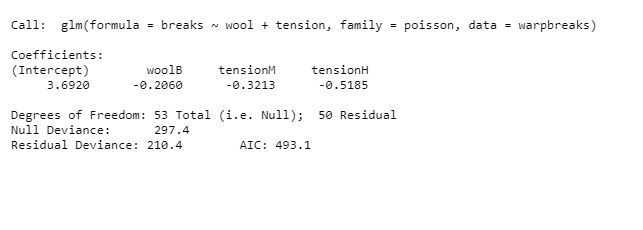
#### Creating Poisson Regression Model using glm() function

**Approach:** Creating the regression model with the help of the **glm()** function as:

* + With the help of this function, easy to make model.
  + Now we draw a graph for the relation between “formula”, “data” and “family”.

**Example:**

|  |
| --- |
| output\_result <-glm(formula = breaks ~ wool + tension,                      data = warpbreaks, family = poisson)  output\_result |

**Output:**  


## What is Non-Linear Regression?

Non-Linear Regression is a statistical method that is used to model the relationship between a dependent variable and one of the independent variable(s). Unlike [linear regression](https://www.geeksforgeeks.org/simple-linear-regression-using-r/), where the relationship between the dependent and independent variables is linear, in non-linear regression, the relationship is modeled using a non-linear equation. This means that the model can capture more complex and non-linear relationships between the variables, but also requires more computational resources and a more sophisticated approach to estimate the model’s parameters. Kindly go through the link for [types of regression](https://www.geeksforgeeks.org/types-of-regression-techniques/)

## What is R?

[R](https://www.geeksforgeeks.org/r-programming-language-introduction/) is an open-source programming language widely used as a statistical software and data analysis tool available across widely used platforms like Windows, MacOS, and Linux. R generally comes with the Command-line interface.

To get started run the following line in the console:

install.packages("minpack.lm")

### Example 1

The code given is of **Exponential regression** in R which uses the **ggplot2** and **nls** libraries. The model is an exponential function. The first is by loading the library and generating some data for the independent variable X and the dependent variable Y. Then, the exponential regression model is fit using the **nls** function. The **nls** function fits a non-linear model to the data using a formula that defines the relationship between the dependent and independent variables. In this case, the formula is **y ~ a \* exp(b \* x)**, which represents an exponential function with parameters a and b. The starting values of these parameters are specified in the start argument with the **list(a=4, b=2).**

#### **The exponential formula is given as**

## R

|  |
| --- |
| # imports library  library(minpack.lm)  library(ggplot2)  # generate data  x <- c(0, 1, 2, 3, 4, 5)  y <- c(1, 2, 4, 8, 16, 32)  # fit the model  start\_values <- c(a=4, b=2)  fit <- nls(y ~ a \* exp(b \* x),             start = start\_values,             algorithm = "port",             control = nls.control(maxiter = 1000))  summary(fit) |

**Output**:

Formula: y ~ a \* exp(b \* x)

Parameters:

Estimate Std. Error t value Pr(>|t|)

a 1.000e+00 6.277e-14 1.593e+13 <2e-16 \*\*\*

b 6.931e-01 1.331e-14 5.206e+13 <2e-16 \*\*\*

---

Signif. codes: 0 ‘\*\*\*’ 0.001 ‘\*\*’ 0.01 ‘\*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 3.247e-13 on 4 degrees of freedom

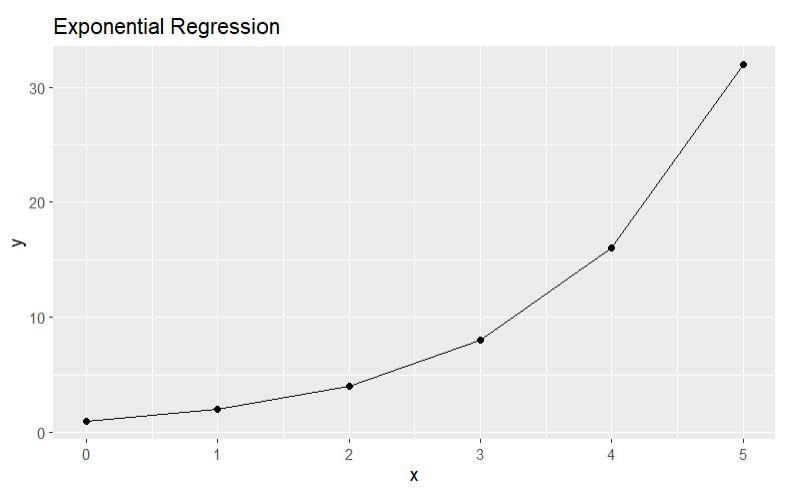
Algorithm "port", convergence message: absolute function convergence (6)

#### **Plot the Exponential Regression line with points**

## R

|  |
| --- |
| # plotting  ggplot(data.frame(x, y), aes(x, y)) +    geom\_point() +    geom\_line(aes(x, predict(fit, newdata = data.frame(x)))) +    ggtitle("Exponential Regression") +    xlab("x") +    ylab("y") |

**Output:**



Exponential Regression model generated using random data

### Example 2

The code given is of **Polynomial regression of degree 2** in R which uses the **ggplot2** and the **lm (linear model)** function from the R library. The data generated has the x variable defined as a sequence of 10 integers (1 to 10) and the y variable is defined as **x2 + x + 2 + random noise**. The **random noise** is generated using the **rnorm** function with a **mean** of 0 and a **standard deviation** of  10. A data frame is created with the **x** and **y** variables and stored in the df variable. The polynomial regression model is fit to the data using the lm function, with the y variable as the response and the **poly(x, 2**) as the predictor.

#### **The polynomial equation is given as**

## R

|  |
| --- |
| #data  x <- 1:10  y <- x^2 + x + 2 + rnorm(10, 0, 10)  df <- data.frame(x, y)  #fitting the model  fit <- lm(y ~ poly(x, 2), data = df)  summary(fit) |

**Output**:

Call:

lm(formula = y ~ poly(x, 2), data = df)

Residuals:

Min 1Q Median 3Q Max

-9.975 -5.664 2.832 5.523 6.885

Coefficients:

Estimate Std. Error t value Pr(>|t|)

(Intercept) 47.140 2.395 19.686 2.18e-07 \*\*\*

poly(x, 2)1 85.981 7.573 11.354 9.21e-06 \*\*\*

poly(x, 2)2 18.278 7.573 2.414 0.0465 \*

---

Signif. codes: 0 ‘\*\*\*’ 0.001 ‘\*\*’ 0.01 ‘\*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 7.573 on 7 degrees of freedom

Multiple R-squared: 0.9506, Adjusted R-squared: 0.9365

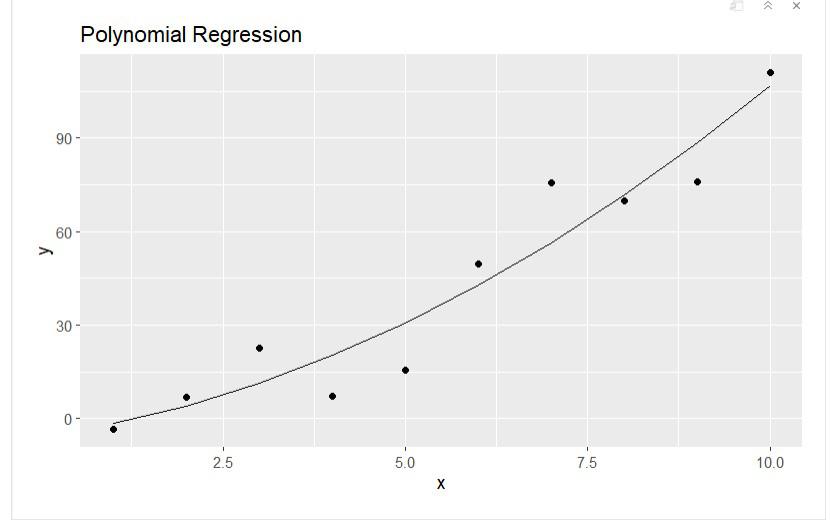
F-statistic: 67.37 on 2 and 7 DF, p-value: 2.676e-05

#### Plot the regression line

## R

|  |
| --- |
| library(ggplot2)  #plotting the model  ggplot(df, aes(x, y)) +    geom\_point() +    geom\_line(aes(x, predict(fit))) +    ggtitle("Polynomial Regression") |

**Output**:



Polynomial Regression generated for numbers 1 to 10

### Example 3

The code given is of **Cubic regression** in R which uses the **ggplot2** and the **lm (linear model)** function from the R library. The data generated has the **x** variable defined as a sequence of 10 integers (1 to 10) and the y variable is defined as  **x3 – 2 x2 + x + 2 + random noise**. The random noise is generated using the **rnorm** function with a mean of 0 and a standard deviation of  10. A data frame is created with the x and y variables and stored in the df variable. The **Cubic regression** model is fit to the data using the lm function.

#### The formula for Cubic equation is given as

## R

|  |
| --- |
| x <- 1:10  y <- x^3 - 2 \* x^2 + x + 2 + rnorm(10, 0, 10)  df <- data.frame(x, y)  fit <- lm(y ~ poly(x, 3), data = df)  summary(fit) |

**Output**:

Call:

lm(formula = y ~ poly(x, 3), data = df)

Residuals:

Min 1Q Median 3Q Max

-11.268 -7.677 -4.150 7.141 16.517

Coefficients:

Estimate Std. Error t value Pr(>|t|)

(Intercept) 234.036 3.826 61.169 1.28e-09 \*\*\*

poly(x, 3)1 748.529 12.099 61.867 1.20e-09 \*\*\*

poly(x, 3)2 328.759 12.099 27.172 1.64e-07 \*\*\*

poly(x, 3)3 61.231 12.099 5.061 0.00231 \*\*

---

Signif. codes: 0 ‘\*\*\*’ 0.001 ‘\*\*’ 0.01 ‘\*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 12.1 on 6 degrees of freedom

Multiple R-squared: 0.9987, Adjusted R-squared: 0.998

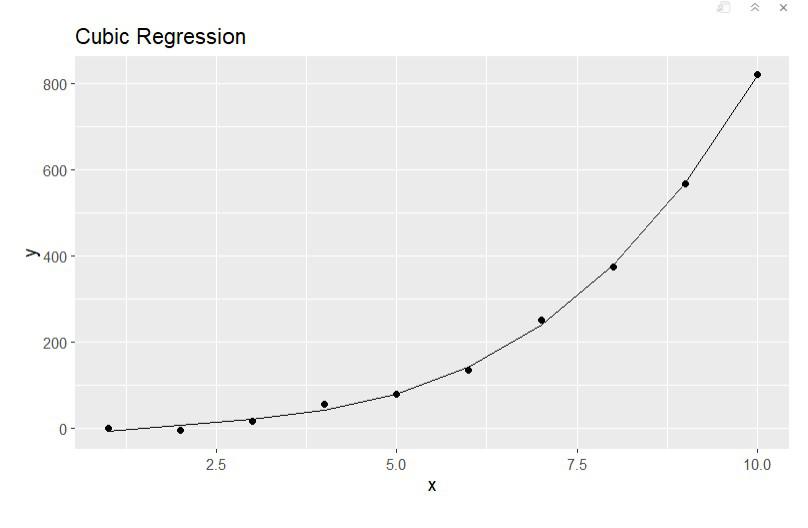
F-statistic: 1530 on 3 and 6 DF, p-value: 4.86e-09

#### Plot the regression line

## R

|  |
| --- |
| #plotting the regression line  library(ggplot2)  ggplot(df, aes(x, y)) +    geom\_point() +    geom\_line(aes(x, predict(fit))) +    ggtitle("Cubic Regression") |

**Output:**



Cubic regression x from 1 to 10

### Example 4

The code given is of **Quadratic regression** in R which uses the **ggplot2** and the **lm (linear model)** function from the R library. The data generated has the x variable defined as a sequence of 10 integers (1 to 10) and the y variable is defined as  **x2 + 2 x + 2 + random noise**. The random noise is generated using the **rnorm** function with a mean of 0 and a standard deviation of  10. A data frame is created with the x and y variables and stored in the df variable. The **quadratic regression** model is fit to the data using the lm function.

#### The Quadratic Formula is

## R

|  |
| --- |
| #x values from 1 to 10  x <- 1:10  #quadratic equation  y <- x^2 + 2 \* x + 2 + rnorm(10, 0, 10)  #creating data frame  df <- data.frame(x, y)  #fitting  the model  fit <- lm(y ~ poly(x, 2), data = df)  summary(fit) |

Output:

Call:

lm(formula = y ~ poly(x, 2), data = df)

Residuals:

Min 1Q Median 3Q Max

-13.2189 -7.5274 0.0416 5.6517 16.7008

Coefficients:

Estimate Std. Error t value Pr(>|t|)

(Intercept) 51.901 3.384 15.339 1.21e-06 \*\*\*

poly(x, 2)1 102.037 10.700 9.536 2.92e-05 \*\*\*

poly(x, 2)2 19.363 10.700 1.810 0.113

---

Signif. codes: 0 ‘\*\*\*’ 0.001 ‘\*\*’ 0.01 ‘\*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 10.7 on 7 degrees of freedom

Multiple R-squared: 0.9308, Adjusted R-squared: 0.9111

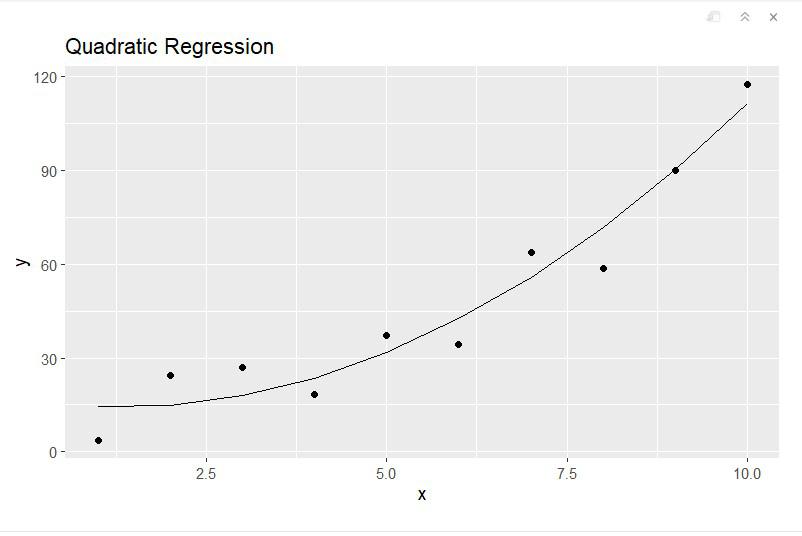
F-statistic: 47.11 on 2 and 7 DF, p-value: 8.699e-05

#### Plot the regression line

## R

|  |
| --- |
| #import libraries  library(ggplot2)  #plotting the model  ggplot(df, aes(x, y)) +    geom\_point() +    geom\_line(aes(x, predict(fit))) +    ggtitle("Quadratic Regression") |

**Output:**



Quadratic equation with x equal to 1 through 10

**Nonparametric regression** is a category of [regression analysis](https://en.wikipedia.org/wiki/Regression_analysis) in which the predictor does not take a predetermined form but is constructed according to information derived from the data. That is, no parametric form is assumed for the relationship between predictors and dependent variable. Nonparametric regression requires larger sample sizes than regression based on [parametric models](https://en.wikipedia.org/wiki/Parametric_model) because the data must supply the model structure as well as the model estimates.

### Nonparametric regression examples

Data for the examples in this chapter are borrowed from the *Correlation and Linear Regression* chapter.  In this hypothetical example, students were surveyed for their weight, daily caloric intake, daily sodium intake, and a score on an assessment of knowledge gain

Data = read.table(header=TRUE, stringsAsFactors=TRUE, text="  
  
Instructor       Grade   Weight  Calories Sodium  Score  
'Brendon Small'     6      43     2069    1287      77  
'Brendon Small'     6      41     1990    1164      76  
'Brendon Small'     6      40     1975    1177      76  
'Brendon Small'     6      44     2116    1262      84  
'Brendon Small'     6      45     2161    1271      86  
'Brendon Small'     6      44     2091    1222      87  
'Brendon Small'     6      48     2236    1377      90  
'Brendon Small'     6      47     2198    1288      78  
'Brendon Small'     6      46     2190    1284      89  
'Jason Penopolis'   7      45     2134    1262      76  
'Jason Penopolis'   7      45     2128    1281      80  
'Jason Penopolis'   7      46     2190    1305      84  
'Jason Penopolis'   7      43     2070    1199      68  
'Jason Penopolis'   7      48     2266    1368      85  
'Jason Penopolis'   7      47     2216    1340      76  
'Jason Penopolis'   7      47     2203    1273      69  
'Jason Penopolis'   7      43     2040    1277      86  
'Jason Penopolis'   7      48     2248    1329      81  
'Melissa Robins'    8      48     2265    1361      67  
'Melissa Robins'    8      46     2184    1268      68  
'Melissa Robins'    8      53     2441    1380      66  
'Melissa Robins'    8      48     2234    1386      65  
'Melissa Robins'    8      52     2403    1408      70  
'Melissa Robins'    8      53     2438    1380      83  
'Melissa Robins'    8      52     2360    1378      74  
'Melissa Robins'    8      51     2344    1413      65  
'Melissa Robins'    8      51     2351    1400      68  
'Paula Small'       9      52     2390    1412      78  
'Paula Small'       9      54     2470    1422      62  
'Paula Small'       9      49     2280    1382      61  
'Paula Small'       9      50     2308    1410      72  
'Paula Small'       9      55     2505    1410      80  
'Paula Small'       9      52     2409    1382      60  
'Paula Small'       9      53     2431    1422      70  
'Paula Small'       9      56     2523    1388      79  
'Paula Small'       9      50     2315    1404      71  
'Coach McGuirk'    10      52     2406    1420      68  
'Coach McGuirk'    10      58     2699    1405      65  
'Coach McGuirk'    10      57     2571    1400      64  
'Coach McGuirk'    10      52     2394    1420      69  
'Coach McGuirk'    10      55     2518    1379      70  
'Coach McGuirk'    10      52     2379    1393      61  
'Coach McGuirk'    10      59     2636    1417      70  
'Coach McGuirk'    10      54     2465    1414      59  
'Coach McGuirk'    10      54     2479    1383      61  
")  
  
  
###  Order factors by the order in data frame  
###  Otherwise, R will alphabetize them  
  
Data$Instructor = factor(Data$Instructor,  
                         levels=unique(Data$Instructor))  
  
  
###  Check the data frame  
  
library(psych)  
  
headTail(Data)  
  
str(Data)  
  
summary(Data)

### Kendall–Theil Sen Siegel nonparametric linear regression

Kendall–Theil regression is a completely nonparametric approach to linear regression where there is one independent and one dependent variable.  It is robust to outliers in the dependent variable.  It simply computes all the lines between each pair of points, and uses the median of the slopes of these lines.  This method is sometimes called Theil–Sen.  A modified, and preferred, method is named after Siegel.

The method yields a slope and intercept for the fit line, and a *p*-value for the slope can be determined as well.  Efron’s pseudo *r-squared* can be determined from the residual and predicted values.

The *mblm* function in the *mblm* package uses the Siegel method by default.  The Theil–Sen procedure can be chosen with the *repeated=FALSE* option. See *library(mblm); ?mblm* for more details.

library(mblm)  
  
model.k = mblm(Calories ~ Sodium,   
               data=Data)  
  
summary(model.k)

Coefficients:  
             Estimate       MAD V value Pr(>|V|)      
(Intercept) -208.5875  608.4540     230 0.000861 \*\*\*  
Sodium         1.8562    0.4381    1035 5.68e-14 \*\*\*  
  
### Values under Estimate are used to determine the fit line.  
### MAD is the median absolute deviation, a robust measure of variability

efronRSquared(model.k)

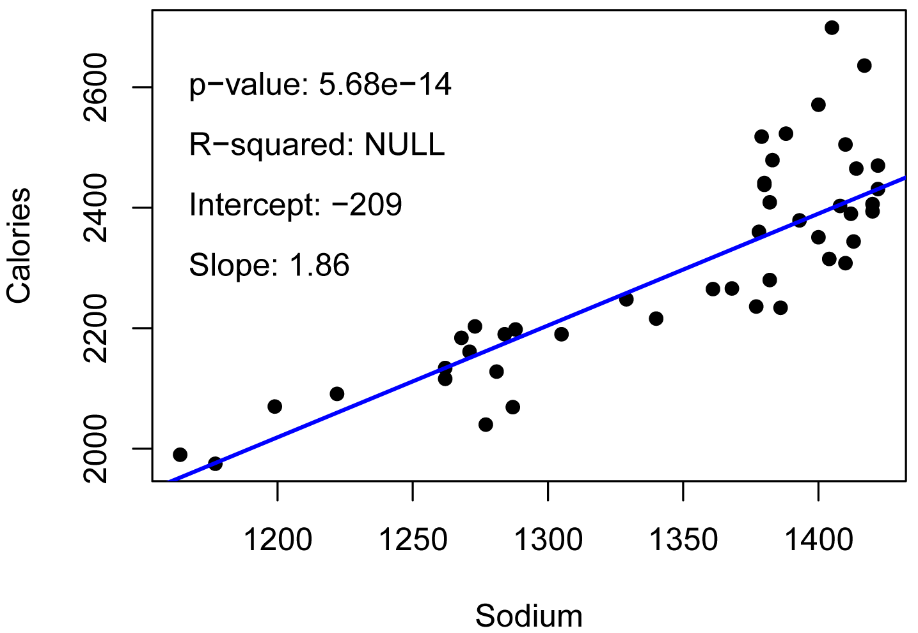
EfronRSquared   
         0.71

accuracy(model.k)

$Fit.criteria  
  Min.max.accuracy  MAE MedAE   MAPE  MSE RMSE NRMSE.mean NRMSE.median Efron.r.squared CV.prcnt  
1            0.972 66.8  48.9 0.0282 8380 91.5     0.0397       0.0397            0.71     3.97

#### Plot with statistics

plot(Calories ~ Sodium,  
     data = Data,  
     pch  = 16)  
  
abline(model.k,  
       col="blue",  
       lwd=2)  
  
Pvalue    = as.numeric(summary(model.k)$coefficients[2,4])  
Intercept = as.numeric(summary(model.k)$coefficients[1,1])  
Slope     = as.numeric(summary(model.k)$coefficients[2,1])  
R2        = NULL  
  
t1     = paste0("p-value: ", signif(Pvalue, digits=3))  
t2     = paste0("R-squared: ", "NULL")  
t3     = paste0("Intercept: ", signif(Intercept, digits=3))  
t4     = paste0("Slope: ", signif(Slope, digits=3))  
  
text(1160, 2600, labels = t1, pos=4)  
text(1160, 2500, labels = t2, pos=4)  
text(1160, 2400, labels = t3, pos=4)  
text(1160, 2300, labels = t4, pos=4)



### Rank-based estimation regression

Rank-based estimation regression uses estimators and inference that are robust to outliers.  It can be used in linear regression situations or in anova-like situations.  The *summary* function from the *Rfit* package produces a type of *r*-squared and a *p*-value for the model.

library(Rfit)  
  
model.r = rfit(Calories ~ Sodium, data = Data)  
  
summary(model.r)

Coefficients:  
              Estimate Std. Error t.value   p.value      
(Intercept) -213.08411  248.42488 -0.8577    0.3958      
Sodium         1.85981    0.18407 10.1036 6.307e-13 \*\*\*  
  
Multiple R-squared (Robust): 0.6637139   
  
Reduction in Dispersion Test: 84.86733 p-value: 0

### Quantile regression

While traditional linear regression models the conditional mean of the dependent variable, quantile regression models the conditional median or other quantile. Medians are most common, but for example, if the factors predicting the highest values of the dependent variable are to be investigated, a 95th percentile could be used.  Likewise, models for several quantiles, e.g. 25th , 50th, 75th percentiles, could be investigated simultaneously.

Quantile regression makes no assumptions about the distribution of the underlying data, and is robust to outliers in the dependent variable.  It does assume the dependent variable is continuous.  However, there are functions for other types of dependent variables in the *qtools* package.  The model assumes that the terms are linearly related. Quantile regression is sometimes considered “semiparametric”.

Quantile regression is very flexible in the number and types of independent variables that can be added to the model.  The example, here, however, confines itself to a simple case with one independent variable and one dependent variable.

This example models the median of dependent variable, which is indicated with the *tau = 0.5* option.

A *p*-value for the model can be found by using the *anova* function with the fit model and the null model.  A pseudo *R-squared* value can be found with the *nagelkerke* function in the *rcompanion* package.

library(quantreg)  
  
model.q = rq(Calories ~ Sodium,  
             data = Data,  
             tau = 0.5)  
  
summary(model.q)

tau: [1] 0.5  
  
Coefficients:  
            coefficients lower bd   upper bd    
(Intercept)  -84.12409   -226.58102  134.91738  
Sodium         1.76642      1.59035    1.89615  
  
### Values under Coefficients are used to determine the fit line.  
### bd appears to be a confidence interval for the coefficients

model.null = rq(Calories ~ 1,  
                data = Data,  
                tau = 0.5)  
  
anova(model.q, model.null)

Quantile Regression Analysis of Deviance Table  
  
  Df Resid Df F value    Pr(>F)      
1  1       43  187.82 < 2.2e-16 \*\*\*  
  
### p-value for model overall

library(rcompanion)  
  
nagelkerke(model.q)

$Pseudo.R.squared.for.model.vs.null  
                             Pseudo.R.squared  
McFadden                             0.115071  
Cox and Snell (ML)                   0.783920  
Nagelkerke (Cragg and Uhler)         0.783921

###### Efron’s pseudo r-squared

library(rcompanion)  
  
accuracy(list(model.q))

   Efron.r.squared  
1            0.707

#### Plot with statistics

plot(Calories ~ Sodium,  
     data = Data,  
     pch  = 16)  
  
abline(model,  
       col="blue",  
       lwd=2)  
  
library(rcompanion)  
  
Pvalue = anova(model.q, model.null)[[1]][1,4]  
Intercept = as.numeric(summary(model.q)$coefficients[1,1])  
Slope     = as.numeric(summary(model.q)$coefficients[2,1])  
R2     = nagelkerke(model.q)[[2]][3,1]  
  
t1     = paste0("p-value: ", signif(Pvalue, digits=3))  
t2     = paste0("R-squared: ", signif(R2, digits=3))  
t3     = paste0("Intercept: ", signif(coefficients(model)[1], digits=3))  
t4     = paste0("Slope: ", signif(coefficients(model)[2], digits=3))  
  
text(1160, 2600, labels = t1, pos=4)  
text(1160, 2500, labels = t2, pos=4)  
text(1160, 2400, labels = t3, pos=4)  
text(1160, 2300, labels = t4, pos=4)

