**Speeding up your R code**

The **apply** functions (**apply**, sapply, **lapply** etc.) are marginally faster than a regular for **loop**, but still do their looping in **R**, rather than dropping down to the lower level of C code. ... On my machine, the vectorised function is >90 times faster than **apply**and >7,000 times faster than the **R loop**!

As an example, here I am just getting the row means of every row of two columns in the following 10000 row dataframe:

N <- 10000

x1 <- runif(N)

x2 <- runif(N)

d <- as.data.frame(cbind(x1, x2))

I compare the time to do this using a for loop…

system.time(for (loop in c(1:length(d[, 1]))) {

d$mean2[loop] <- mean(c(d[loop, 1], d[loop, 2]))

})

## user system elapsed

## 13.912 0.204 14.150

the apply function…

system.time(d$mean1 <- apply(d, 1, mean))

## user system elapsed

## 0.180 0.000 0.179

and the vectorised rowMeans function:

system.time(d$mean3 <- rowMeans(d[, c(1, 2)]))

## user system elapsed

## 0.004 0.000 0.002

dplyr is a new package which provides a set of tools for efficiently manipulating datasets in R. dplyr is the next iteration of plyr, focussing on only data frames. dplyr is faster, has a more consistent API and should be easier to use. There are three key ideas that underlie dplyr:

1. Your time is important, so Romain Francois has written the key pieces in Rcpp to provide blazing fast performance. Performance will only get better over time, especially once we figure out the best way to make the most of multiple processors.
2. Tabular data is tabular data regardless of where it lives, so you should use the same functions to work with it. With dplyr, anything you can do to a local data frame you can also do to a remote database table. PostgreSQL, MySQL, SQLite and Google bigquery support is built-in; adding a new backend is a matter of implementing a handful of S3 methods.
3. The bottleneck in most data analyses is the time it takes for you to figure out what to do with your data, and dplyr makes this easier by having individual functions that correspond to the most common operations (group\_by, summarise, mutate, filter, select and arrange). Each function does one only thing, but does it well.

In plyr, we might write code like this:

**library**(Lahman)

**library**(plyr)

games <- ddply(Batting, "playerID", summarise, total = sum(G))

head(arrange(games, desc(total)), 5)

We use ddply() to break up the Batting dataframe into pieces according to the playerID variable, then apply summarise() to reduce the player data to a single row. Each row in Batting represents one year of data for one player, so we figure out the total number of games with sum(G) and save it in a new variable called total. We sort the result so the most games come at the top and then use head() to pull off the first five.

In dplyr, the code is similar:

**library**(Lahman)

**library**(dplyr)

players <- group\_by(Batting, playerID)

games <- summarise(players, total = sum(G))

head(arrange(games, desc(total)), 5)

## **What is the Random Forest algorithm?**

Random forest is a tree-based algorithm which involves building several trees (decision trees), then combining their output to improve generalization ability of the model. The method of combining trees is known as an ensemble method. Ensembling is nothing but a combination of weak learners (individual trees) to produce a strong learner.

Say, you want to watch a movie. But you are uncertain of its reviews. You ask 10 people who have watched the movie. 8 of them said " the movie is fantastic." Since the majority is in favor, you decide to watch the movie. This is how we use ensemble techniques in our daily life too.

Random Forest can be used to solve regression and classification problems. In regression problems, the dependent variable is continuous. In classification problems, the dependent variable is categorical.

1. Given a data frame (n x p), a tree stratifies or partitions the data based on rules (if-else). Yes, a tree creates rules. These rules divide the data set into distinct and non-overlapping regions. These rules are determined by a variable's contribution to the homogenity or pureness of the resultant child nodes (X2,X3).

2. In the image above, the variable X1 resulted in highest homogeneity in child nodes, hence it became the root node. A variable at root node is also seen as the most important variable in the data set.

3, But how is this homogeneity or pureness determined? In other words, how does the tree decide at which variable to split?

* In **regression trees** (where the output is predicted using the mean of observations in the terminal nodes), the splitting decision is based on minimizing RSS. The variable which leads to the greatest possible reduction in RSS is chosen as the root node. The tree splitting takes a **top-down greedy** approach, also known as *recursive binary splitting*. We call it "greedy" because the algorithm cares to make the best split at the current step rather than saving a split for better results on future nodes.
* In **classification trees** (where the output is predicted using mode of observations in the terminal nodes), the splitting decision is based on the following methods:
  + **Gini Index** - It's a measure of node purity. If the Gini index takes on a smaller value, it suggests that the node is pure. For a split to take place, the Gini index for a child node should be less than that for the parent node.
  + **Entropy** - Entropy is a measure of node impurity. For a binary class (a,b), the formula to calculate it is shown below. Entropy is maximum at p = 0.5. For p(X=a)=0.5 or p(X=b)=0.5 means, a new observation has a 50%-50% chance of getting classified in either classes. The entropy is minimum when the probability is 0 or 1.

## Advantages and Disadvantages of Random Forest

Advantages are as follows:

1. It is robust to correlated predictors.
2. It is used to solve both regression and classification problems.
3. It can be also used to solve unsupervised ML problems.
4. It can handle thousands of input variables without variable selection.
5. It can be used as a feature selection tool using its variable importance plot.
6. It takes care of missing data internally in an effective manner.

Disadvantages are as follows:

1. The Random Forest model is difficult to interpret.
2. It tends to return erratic predictions for observations out of range of training data. For example, the training data contains two variable x and  y. The range of x variable is 30 to 70. If the test data has x = 200, random forest would give an unreliable prediction.
3. It can take longer than expected time to computer a large number of trees.

https://archive.ics.uci.edu/ml/machine-learning-databases/adult/

## Solving a Problem (Parameter Tuning)

Let's take a data set to compare the performance of bagging and random forest algorithms. Along the way, I'll also explain important parameters used for parameter tuning. In R, we'll use MLR and data.table package to do this analysis. I've taken the Adult dataset from the UCI machine learning repository. You can download the data from here.

This data set presents a binary classification problem to solve. Given a set of features, we need to predict if a person's salary is <=50K or >=50k. Since the given data isn't well structured, we'll need to make some modification while reading the data set.

#set working directory

path <- "~/December 2016/RF\_Tutorial"

setwd(path)

#load libraries

library(data.table)

library(mlr)

library(h2o)

#set variable names

setcol <- c("age",` `"workclass",` `"fnlwgt",` `"education",` `"education-num",` `"marital-status",` `"occupation",` `"relationship",` `"race",` `"sex",` `"capital-gain",` `"capital-loss",` `"hours-per-week",` `"native-country",` `"target")

#load data

train <- read.table("adultdata.txt",header = F,sep = ",",col.names = setcol,na.strings = c(" ?"),stringsAsFactors = F)

test <- read.table("adulttest.txt",header = F,sep = ",",col.names = setcol,skip = 1, na.strings = c(" ?"),stringsAsFactors = F)

After we've loaded the data set, first we'll set the data class to data.table. data.table is the most powerful R package made for faster data manipulation.

setDT(train)

setDT(test)

Now, we'll quickly look at given variables, data dimensions, etc.

dim(train)

dim(test)

str(train)

str(test)

As seen from the output above, we can derive the following insights:

1. The train data set has 32,561 rows and 15 columns.
2. The test data has 16,281 rows and 15 columns.
3. Variable target is the dependent variable.
4. The target variable in train and test data is different. We'll need to match them.
5. All character variables have a leading whitespace which can be removed.

We can check missing values using:

#check missing values

table(is.na(train))

FALSE   TRUE

484153  4262

sapply(train, function(x) sum(is.na(x))/length(x))\*100

table(is.na(test))

FALSE  TRUE

242012 2203

sapply(test, function(x) sum(is.na(x))/length(x))\*100`<br/>

As seen above, both train and test datasets have missing values. The sapply function is quite handy when it comes to performing column computations. Above, it returns the percentage of missing values per column. Now, we'll preprocess the data to prepare it for training. In R, random forest internally takes care of missing values using mean/ mode imputation. Practically speaking, sometimes it takes longer than expected for the model to run.

Therefore, in order to avoid waiting time, let's impute the missing values using median / mode imputation method; i.e. missing values in the integer variable will be imputed with median and factor variables will be imputed with mode (most frequent value). We'll use the impute function from MLR package, which is enabled with several unique methods for missing value imputation:

imp1 <- impute(data = train,target = "target",classes = list(integer=imputeMedian(), factor=imputeMode()))

imp2 <- impute(data = test,target = "target",classes = list(integer=imputeMedian(), factor=imputeMode()))

train <- imp1$data

test <- imp2$data

Being a binary classification problem, you are always advised to check if the data is imbalanced or not. We can do it in the following way:

setDT(train)[,.N/nrow(train),target]

target     V1

1: <=50K   0.7591904

2: >50K    0.2408096

setDT(test)[,.N/nrow(test),target]

target     V1

1: <=50K.  0.7637737

2: >50K.   0.2362263

If you observe carefully, the value of the target variable is different in test and train. For now, we can consider it  a typo error and correct all the test values. Also, we see that 75% of people in train data have income <=50K. Imbalanced classification problems are known to be more skewed with a binary class distribution of 90% to 10%. Now, let's proceed and clean the target column in test data.

test[,target := substr(target,start = 1,stop = nchar(target)-1)]

We've used the substr function to return the subtring from a specified start and end position. Next, we'll remove the leading whitespaces from all character variables. We'll use str\_trim function from stringr package.

library(stringr)

char\_col <- colnames(train)[sapply(train,is.character)]

for(i in char\_col)

set(train,j=i,value = str\_trim(train[[i]],side = "left"))

Using sapply function, we've extracted the column names which have character class. Then, using a simple for - set loop we traversed all those columns and applied the str\_trim function. Before we start model training, we should convert all character variables to factor. MLR package treats character class as unknown.

fact\_col <- colnames(train)[sapply(train,is.character)]

for(i in fact\_col)

set(train,j=i,value = factor(train[[i]]))

for(i in fact\_col)

set(test,j=i,value = factor(test[[i]]))

Let's start with modeling now. MLR package has its own function to convert data into a task, build learners, and optimize learning algorithms. I suggest you stick to the modeling structure described below for using MLR on any data set.

#create a task

traintask <- makeClassifTask(data = train,target = "target")

testtask <- makeClassifTask(data = test,target = "target")

#create learner

bag <- makeLearner("classif.rpart",predict.type = "response")

bag.lrn <- makeBaggingWrapper(learner = bag,bw.iters = 100,bw.replace = TRUE)

I've set up the bagging algorithm which will grow 100 trees on randomized samples of data with replacement. To check the performance, let's set up a validation strategy too:

#set 5 fold cross validation

rdesc <- makeResampleDesc("CV",iters=5L)

For faster computation, we'll use parallel computation backend. Make sure your machine / laptop doesn't have many programs running at backend.

#set parallel backend (Windows)

library(parallelMap)

library(parallel)

parallelStartSocket(cpus = detectCores())

For linux users, the function parallelStartMulticore(cpus = detectCores()) will activate parallel backend. I've used all the cores here.

r <- resample(learner = bag.lrn , task = traintask, resampling = rdesc, measures = list(tpr,fpr,fnr,fpr,acc) ,show.info = T)

#[Resample] Result:

# tpr.test.mean=0.95 #fnr.test.mean=0.0505 #fpr.test.mean=0.487 #acc.test.mean=0.845

Being a binary classification problem, I've used the components of confusion matrix to check the model's accuracy. With 100 trees, bagging has returned an accuracy of 84.5%, which is way better than the baseline accuracy of 75%. Let's now check the performance of  random forest.

#make randomForest learner

rf.lrn <- makeLearner("classif.randomForest")

rf.lrn$par.vals <- list(ntree = 100L, importance=TRUE)

r <- resample(learner = rf.lrn, task = traintask, resampling = rdesc, measures = list(tpr,fpr,fnr,fpr,acc), show.info = T)

# Result:

# tpr.test.mean=0.996 #fpr.test.mean=0.72 #fnr.test.mean=0.0034 #acc.test.mean=0.825

On this data set, random forest performs worse than bagging. Both used 100 trees and random forest returns an overall accuracy of 82.5 %. An apparent reason being that this algorithm is messing up classifying the negative class. As you can see, it classified 99.6% of the positive classes correctly, which is way better than the bagging algorithm. But it incorrectly classified 72% of the negative classes.

Internally, random forest uses a cutoff of 0.5; i.e., if a particular unseen observation has a probability higher than 0.5, it will be classified as <=50K. In random forest, we have the option to customize the internal cutoff. As the false positive rate is very high now, we'll increase the cutoff for positive classes (<=50K) and accordingly reduce it for negative classes (>=50K). Then, train the model again.

#set cutoff

rf.lrn$par.vals <- list(ntree = 100L, importance=TRUE, cutoff = c(0.75,0.25))

r <- resample(learner = rf.lrn, task = traintask, resampling = rdesc, measures = list(tpr,fpr,fnr,fpr,acc), show.info = T)

#Result: tpr.test.mean=0.934 #fpr.test.mean=0.43 #fnr.test.mean=0.0662 #acc.test.mean=0.846

As you can see, we've improved the accuracy of the random forest model by 2%, which is slightly higher than that for the bagging model. Now, let's try and make this model better.

**Parameter Tuning:** Mainly, there are three parameters in the random forest algorithm which you should look at (for tuning):

* **ntree** - As the name suggests, the number of trees to grow. Larger the tree, it will be more computationally expensive to build models.
* **mtry** - It refers to how many variables we should select at a node split. Also as mentioned above, the default value is p/3 for regression and sqrt(p) for classification. We should always try to avoid using smaller values of mtry to avoid overfitting.
* **nodesize** - It refers to how many observations we want in the terminal nodes. This parameter is directly related to tree depth. Higher the number, lower the tree depth. With lower tree depth, the tree might even fail to recognize useful signals from the data.

Let get to the playground and try to improve our model's accuracy further. In MLR package, you can list all tuning parameters a model can support using:

getParamSet(rf.lrn)

#set parameter space

params <- makeParamSet(makeIntegerParam("mtry",lower = 2,upper = 10),makeIntegerParam("nodesize",lower = 10,upper = 50))

#set validation strategy

rdesc <- makeResampleDesc("CV",iters=5L)

#set optimization technique

ctrl <- makeTuneControlRandom(maxit = 5L)

#start tuning

tune <- tuneParams(learner = rf.lrn, task = traintask, resampling = rdesc, measures = list(acc), par.set = params, control = ctrl, show.info = T)

#[Tune] Result: mtry=2 : nodesize=23 : acc.test.mean=0.858

After tuning, we have achieved an overall accuracy of 85.8%, which is better than our previous random forest model. This way you can tweak your model and improve its accuracy.

# Hierarchical Clustering

## **What is hierarchical clustering?**

If you recall from the post about k means clustering, it requires us to specify the number of clusters, and finding the optimal number of clusters can often be hard. Hierarchical clustering is an alternative approach which builds a hierarchy from the bottom-up, and doesn’t require us to specify the number of clusters beforehand.

The algorithm works as follows:

* Put each data point in its own cluster.
* Identify the closest two clusters and combine them into one cluster.
* Repeat the above step till all the data points are in a single cluster.

Once this is done, it is usually represented by a dendrogram like structure.

There are a few ways to determine how close two clusters are:

* Complete linkage clustering: Find the maximum possible distance between points belonging to two different clusters.
* Single linkage clustering: Find the minimum possible distance between points belonging to two different clusters.
* Mean linkage clustering: Find all possible pairwise distances for points belonging to two different clusters and then calculate the average.
* Centroid linkage clustering: Find the centroid of each cluster and calculate the distance between centroids of two clusters.

Complete linkage and mean linkage clustering are the ones used most often.

## **Clustering**

In my post on K Means Clustering, we saw that there were 3 different species of flowers.

Let us see how well the hierarchical clustering algorithm can do. We can use hclust for this. hclust requires us to provide the data in the form of a distance matrix. We can do this by using dist. By default, the complete linkage method is used.

clusters <- hclust(dist(iris[, 3:4]))

plot(clusters)

clusterCut <- cutree(clusters, 3)

Now, let us compare it with the original species.

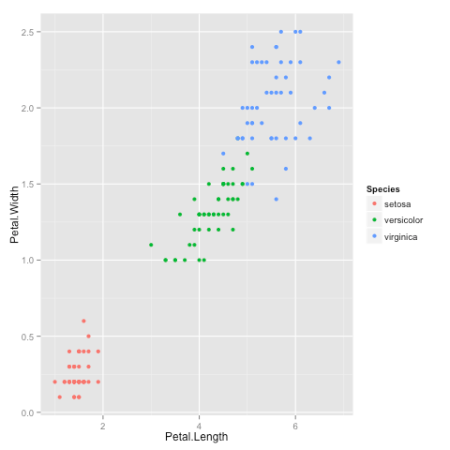
table(clusterCut, iris$Species)

*clusterCut setosa versicolor virginica*

*1 50 0 0*

*2 0 21 50*

*3 0 29 0*

It looks like the algorithm successfully classified all the flowers of species setosa into cluster 1, and virginica into cluster 2, but had trouble with versicolor. If you look at the original plot showing the different species, you can understand why:[](https://i2.wp.com/datascienceplus.com/wp-content/uploads/2015/12/plot1.png)

Let us see if we can better by using a different linkage method. This time, we will use the mean linkage method:

clusters <- hclust(dist(iris[, 3:4]), method = 'average')

plot(clusters)

## **What is K Means Clustering?**

K Means Clustering is an unsupervised learning algorithm that tries to cluster data based on their similarity. Unsupervised learning means that there is no outcome to be predicted, and the algorithm just tries to find patterns in the data. In k means clustering, we have the specify the number of clusters we want the data to be grouped into. The algorithm randomly assigns each observation to a cluster, and finds the centroid of each cluster. Then, the algorithm iterates through two steps:

* Reassign data points to the cluster whose centroid is closest.
* Calculate new centroid of each cluster.

These two steps are repeated till the within cluster variation cannot be reduced any further. The within cluster variation is calculated as the sum of the euclidean distance between the data points and their respective cluster centroids.

## **Exploring the data**

The iris dataset contains data about sepal length, sepal width, petal length, and petal width of flowers of different species. Let us see what it looks like:

library(datasets)

head(iris)

Sepal.Length Sepal.Width Petal.Length Petal.Width Species

1 5.1 3.5 1.4 0.2 setosa

2 4.9 3.0 1.4 0.2 setosa

3 4.7 3.2 1.3 0.2 setosa

4 4.6 3.1 1.5 0.2 setosa

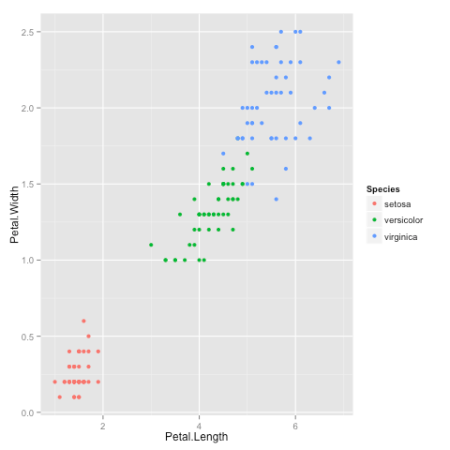
5 5.0 3.6 1.4 0.2 setosa

6 5.4 3.9 1.7 0.4 setosa

After a little bit of exploration, I found that Petal.Length and Petal.Widthwere similar among the same species but varied considerably between different species, as demonstrated below:

library(ggplot2)

ggplot(iris, aes(Petal.Length, Petal.Width, color = Species)) + geom\_point()

Here is the plot:  
[](https://i2.wp.com/datascienceplus.com/wp-content/uploads/2015/12/plot1.png)

## **Clustering**

Okay, now that we have seen the data, let us try to cluster it. Since the initial cluster assignments are random, let us set the seed to ensure reproducibility.

set.seed(20)

irisCluster <- kmeans(iris[, 3:4], 3, nstart = 20)

irisCluster

K-means clustering with 3 clusters of sizes 46, 54, 50

Cluster means:

Petal.Length Petal.Width

1 5.626087 2.047826

2 4.292593 1.359259

3 1.462000 0.246000

Clustering vector:

[1] 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3

[35] 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2

[69] 2 2 2 2 2 2 2 2 2 1 2 2 2 2 2 1 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 1 1

[103] 1 1 1 1 2 1 1 1 1 1 1 1 1 1 1 1 1 2 1 1 1 2 1 1 2 2 1 1 1 1 1 1 1 1

[137] 1 1 2 1 1 1 1 1 1 1 1 1 1 1

Within cluster sum of squares by cluster:

[1] 15.16348 14.22741 2.02200

(between\_SS / total\_SS = 94.3 %)

Available components:

[1] "cluster" "centers" "totss" "withinss"

[5] "tot.withinss" "betweenss" "size" "iter"

[9] "ifault"

Since we know that there are 3 species involved, we ask the algorithm to group the data into 3 clusters, and since the starting assignments are random, we specify nstart = 20. This means that R will try 20 different random starting assignments and then select the one with the lowest within cluster variation.  
We can see the cluster centroids, the clusters that each data point was assigned to, and the within cluster variation.

Let us compare the clusters with the species.

table(irisCluster$cluster, iris$Species)

setosa versicolor virginica

1 0 2 44

2 0 48 6

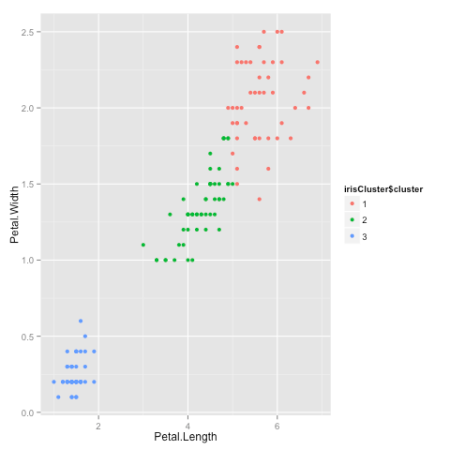
3 50 0 0

As we can see, the data belonging to the setosa species got grouped into cluster 3, versicolor into cluster 2, and virginica into cluster 1. The algorithm wrongly classified two data points belonging to versicolor and six data points belonging to virginica.

We can also plot the data to see the clusters:

irisCluster$cluster <- as.factor(irisCluster$cluster)

ggplot(iris, aes(Petal.Length, Petal.Width, color = iris$cluster)) + geom\_point()

Here is the plot:  
[](https://i2.wp.com/datascienceplus.com/wp-content/uploads/2015/12/plot2.png)

**R nonlinear regression**

Performing **Nonlinear** Least Square and **Nonlinear** Regressions in **R**. Linear**regression** is a basic tool. It works on the assumption that there exists a linear relationship between the dependent and independent variable, also known as the explanatory variables and output.

Nls()

we observe that it is rarely the case that the equation of the model is a linear equation giving a linear graph. Most of the time, the equation of the model of real world data involves mathematical functions of higher degree like an exponent of 3 or a sin function. In such a scenario, the plot of the model gives a curve rather than a line. The goal of both linear and non-linear regression is to adjust the values of the model's parameters to find the line or curve that comes closest to your data. On finding these values we will be able to estimate the response variable with good accuracy.

In Least Square regression, we establish a regression model in which the sum of the squares of the vertical distances of different points from the regression curve is minimized. We generally start with a defined model and assume some values for the coefficients. We then apply the **nls()** function of R to get the more accurate values along with the confidence intervals.

## **Syntax**

The basic syntax for creating a nonlinear least square test in R is −

nls(formula, data, start)

Following is the description of the parameters used −

* **formula** is a nonlinear model formula including variables and parameters.
* **data** is a data frame used to evaluate the variables in the formula.
* **start** is a named list or named numeric vector of starting estimates.

## **Example**

We will consider a nonlinear model with assumption of initial values of its coefficients. Next we will see what is the confidence intervals of these assumed values so that we can judge how well these values fir into the model.

So let's consider the below equation for this purpose −

a = b1\*x^2+b2

Let's assume the initial coefficients to be 1 and 3 and fit these values into nls() function

xvalues <- c(1.6,2.1,2,2.23,3.71,3.25,3.4,3.86,1.19,2.21)

yvalues <- c(5.19,7.43,6.94,8.11,18.75,14.88,16.06,19.12,3.21,7.58)

# Give the chart file a name.

png(file = "nls.png")

# Plot these values.

plot(xvalues,yvalues)

# Take the assumed values and fit into the model.

model <- nls(yvalues ~ b1\*xvalues^2+b2,start = list(b1 = 1,b2 = 3))

# Plot the chart with new data by fitting it to a prediction from 100 data points.

new.data <- data.frame(xvalues = seq(min(xvalues),max(xvalues),len = 100))

lines(new.data$xvalues,predict(model,newdata = new.data))

# Save the file.

dev.off()

# Get the sum of the squared residuals.

print(sum(resid(model)^2))

# Get the confidence intervals on the chosen values of the coefficients.

print(confint(model))

When we execute the above code, it produces the following result −

[1] 1.081935

Waiting for profiling to be done...

2.5% 97.5%

b1 1.137708 1.253135

b2 1.497364 2.496484