Data wrangling : Transforming (2/3)

Data wrangling is a task of great importance in data analysis. Data wrangling, is the process of importing, cleaning and transforming raw data into actionable information for analysis. It is a time-consuming process which is estimated to take about 60-80% of analyst's time. In this series we will go through this process. It will be a brief series with goal to craft the reader's skills on the data wrangling task. This is the third part of the series and it aims to cover the transforming of data used.This can include filtering, summarizing, and ordering your data by different means. This also includes combining various data sets, creating new variables, and many other manipulation tasks. At this post, we will go through a few more advanced transformation tasks on mtcars data set.

Before proceeding, it might be helpful to look over the help pages for the group_by, ungrpoup, summary, summarise, arrange, mutate, cumsum.

Moreover please load the following libraries.
install.packages("dplyr")
library(dplyr)

Answers to the exercises are available here.

If you obtained a different (correct) answer than those listed on the solutions page, please feel free to post your answer as a comment on that page.

Exercise 1

Create a new object named *cars_cyl* and assign to it the *mtcars* data frame grouped by the variable *cyl*
Hint: be careful about the data type of the variable, in order to be used for grouping it has to be a factor.

Exercise 2

Remove the grouping from the object *cars_cyl*

Exercise 3

Print out the summary statistics of the *mtcars* data frame using the summary function and pipeline symbols %>%.

Learn more about Data Pre-Processing in the online course R Data Pre-Processing & Data Management – Shape your Data!. In this course you will learn how to:

- Work with popular libraries such as dplyr
- Learn about methods such as pipelines
- And much more

Exercise 4

Make a more descriptive summary statistics output containing the 4 quantiles, the mean, the standard deviation and the count.

Exercise 5

Print out the average *hp* for every *cyl* category

Exercise 6

Print out the *mtcars* data frame sorted by *hp* (ascending oder)

Exercise 7

Print out the *mtcars* data frame sorted by *hp* (descending oder)

Exercise 8

Create a new object named *cars_per* containing the *mtcars* data frame along with a new variable called *performance* and calculated as performance = hp/mpg

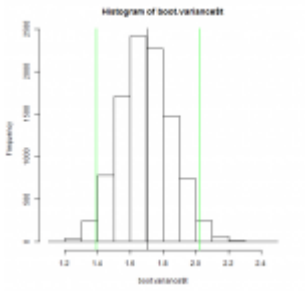
Exercise 9

Print out the cars_per data frame, sorted by *performance* in descending order and create a new variable called *rank* indicating the rank of the cars in terms of performance.

Exercise 10

To wrap everything up, we will use the *iris* data set. Print out the mean of every variable for every *Species* and create two new variables called *Sepal.Density* and *Petal.Density* being calculated as Sepal.Density = Sepal.Length Sepal.Width and Petal.Density = Sepal.Length Petal.Width respectively.

Hacking statistics or: How I Learned to Stop Worrying About Calculus and Love Stats Exercises (Part-3)



Statistics are often taught in school by and for people who like Mathematics. As a consequence, in those class emphasis is put on leaning equations, solving calculus problems and creating mathematics models instead of building an intuition for probabilistic problems. But, if you read this, you know a

bit of R programming and have access to a computer that is really good at computing stuff! So let's learn how we can tackle useful statistic problems by writing simple R query and how to think in probabilistic terms.

In the first two part of this series, we've seen how to identify the distribution of a random variable by plotting the distribution of a sample and by estimating statistic. We also seen that it can be tricky to identify a distribution from a small sample of data. Today, we'll see how to estimate the confidence interval of a statistic in this situation by using a powerful method called bootstrapping.

Answers to the exercises are available here.

Exercise 1
Load this dataset and draw the histogram, the ECDF of this sample and the ECDF of a density who's a good fit for the data.

Exercise 2
Write a function that takes a dataset and a number of iterations as parameter. For each iteration this function must create a sample with replacement of the same size than the dataset, calculate the mean of the sample and store it in a matrix, which the function must return.

Exercise 3
Use the t.test() to compute the 95% confidence interval estimate for the mean of your dataset.

- And much more

Exercise 4
Use the function you just wrote to estimate the mean of your sample 10,000 times. Then draw the histogram of the results and the sampling mean of the data.

The probability distribution of the estimation of a mean is a normal distribution centered around the real value of the mean. In other words, if we take a lot of samples from a population and compute the mean of each sample, the histogram of those mean will look like one of a normal distribution center around the real value of the mean we try to estimate. We have recreated artificially this process by creating a bunch of new sample from the dataset, by resampling it with replacement and now we can do a point estimation of the mean by computing the average of the sample of means or compute the confidence interval by finding the correct percentile of this distribution. This process is basically what is called bootstrapping.

Exercise 5
Calculate the value of the 2.5 and 97.5 percentile of your sample of 10,000 estimates of the mean and the mean of this sample. Compare this last value to the value of the sample mean of your data.

Exercise 6
Bootstrapping can be used to compute the confidence interval of all the statistics of interest, but you don’t have to write a function for each of them! You can use the boot() function from the library of the same name and pass the statistic as argument to compute the bootstrapped sample. Use this function with 10,000 replicates to compute the median of the dataset.

Exercise 7
Look at the structure of your result and plot his histogram. On the same plot, draw the value of the sample median of your

dataset and plot the 95% confidence interval of this statistic by adding two vertical green lines at the lower and higher bounds of the interval.

Exercise 8
Write functions to compute by bootstrapping the following statistics:

- Variance
- kurtosis
- Max
- Min

Exercise 9
Use the functions from last exercise and the boot function with 10,000 replicates to compute the following statistics:

- Variance
- kurtosis
- Max
- Min

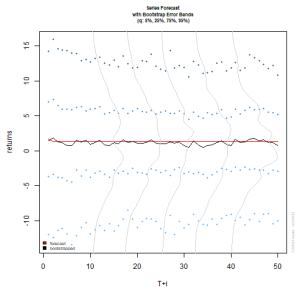
Then draw the histogram of the bootstrapped sample and plot the 95% confidence interval of the statistics.

Exercise 10
Generate 1000 points from a normal distribution of mean and standard deviation equal to the one of the dataset. Use the bootstrap method to estimate the 95% confidence interval of the mean, the variance, the kurtosis, the min and the max of this density. Then plot the histograms of the bootstrap samples for each of the variable and draw the 95% confidence interval as two red vertical line.

Two bootstrap estimate of the same statistic of two sample who are distributed by the same density should be pretty similar. When we compare those last plots with the confidence interval we drawn before we see that they are. More importantly, the confidence interval computed in exercise 10 overlap the

confidence interval of the statistics of the first dataset. As a consequence, we can't conclude that the two sample come from different density distribution and in practice we could use a normal distribution with a mean of 0.4725156 and a standard deviation of 1.306665 to simulate this random variable.

Volatility modelling in R exercises (Part-4)



This is the fourth part of the series on volatility modelling. For other parts of the series follow the tag volatility.

In this exercise set we will explore GARCH-M and E-GARCH models. We will also use these models to generate rolling window forecasts, bootstrap forecasts and perform simulations.

Answers to the exercises are available here.

Exercise 1
Load the rugarch and the FinTS packages. Next, load the m.ibmspln dataset from the FinTS package. This dataset contains monthly excess returns of the S&P500 index and IBM stock from Jan-1926 to Dec-1999 (Ruey Tsay (2005) Analysis of Financial Time Series, 2nd ed. ,Wiley, chapter 3).
Also, load the forecast package which we will use for autocorrelation graphs.

Exercise 2

Estimate a GARCH(1,1)-M model for the S&P500 excess returns series. Determine if the effect of volatility on asset returns is significant.

Exercise 3

Excess IBM stock returns are defined as a regular zoo variable. Convert this to a time series variable with correct dates.

Learn more about Model Evaluation in the online course Regression Machine Learning with R. In this course you will learn how to:

- Avoid model over-fitting using cross-validation for optimal parameter selection
- Explore maximum margin methods such as best penalty of error term support vector machines with linear and non-linear kernels.
- And much more

Exercise 4

Plot the absolute and squared excess IBM stock returns along with its ACF and PACF graphs and determine the appropriate model configuration.

Exercise 5

The exponential GARCH model incorporates asymmetric effects for positive and negative asset returns. Estimate an AR(1)-EGARCH(1,1) model for the IBM series.

Exercise 6

Using the results from exercise-5, get rolling window forecasts starting from the 800th observation and refit the model after every three observations.

Exercise 7

Estimate an AR(1)-GARCH(1,1) model for the IBM series and get a bootstrap forecast for the next 50 periods with 500 replications.

Exercise 8
Plot the forecasted returns and sigma with bootstrap error bands.

Exercise 9
We can use Monte-Carlo simulation to get a distribution of the parameter estimates. Using the fitted model from exercise-7, run the simulation for 500 periods for a horizon of 2000 periods.

Exercise 10
Plot the density functions of the parameter estimates.

Data visualization with googleVis exercises part 7

Table, Org Chart & Tree Map

In the seventh part of our series we are going to learn about the features of some interesting types of charts. More specifically we will talk about Table, Org Chart and Tree Map.

Read the examples below to understand the logic of what we are going to do and then test yous skills with the exercise set we

prepared for you. Lets begin!

Answers to the exercises are available here.

Package

As you already know, the first thing you have to do is install and load the googleVis package with:

```
install.packages("googleVis")
library(googleVis)
```

NOTE: The charts are created locally by your browser. In case they are not displayed at once press F5 to reload the page.

Table

It is quite simple to create a Table with googleVis. We will use the "Stock" dataset.
Look at the example below to create a simple table:

```
TableC <- gvisTable(Stock)
plot(TableC)
```

Exercise 1

Create a list named "TableC" and pass to it the "Stock" dataset as a table. **HINT**: Use gvisTable().

Exercise 2

Plot the the table. **HINT**: Use plot().

Table with pages

To add pages to your table use:

```
options=list(page='enable')
```

Exercise 3

Add pages to the table you just created and plot it. **HINT**: Use list().

Org chart

It is quite simple to create an Org Chart with googleVis. We will use the “Regions” dataset. You can see the variables of your dataset with head().
Look at the example below to create a simple Org Chart:

```
OrgC <- gvisOrgChart(Regions )
plot(OrgC)
```

Learn more about using GoogleVis in the online course Mastering in Visualization with R programming. In this course you will learn how to:

- Work extensively with the GoogleVis package and its functionality
- Learn what visualizations exist for your specific use case
- And much more

Exercise 4

Create a list named “OrgC” and pass to it the “Regions” dataset as an org chart. **HINT**: Use gvisOrgChart().

Exercise 5

Plot the the org chart. **HINT**: Use plot().

Dimensions

You can adjust the dimensions of the org chart with these options:

```
options=list(width=600, height=250,
size='large')
```

Exercise 6

Adjust the dimensions of your org chart. Set height to 300, width to 550 and size to medium and plot it.

Tree Map

It is quite simple to create a Tree Map with googleVis. We will use the “Regions” dataset.
Look at the example below to create a simple Tree Map:

```
TreeC <- gvisTreeMap(Regions)
plot(TreeC)
```

Exercise 7

Create a list named “TreeC” and pass to it the “Regions” dataset as an org chart. **HINT**: Use gvisTreeMap().

Exercise 8

Plot the the tree map. **HINT**: Use plot().

You can decide tha dependents variables of your dataset by selecting it. In the example above the dependent variable was “Val”. To choose “Fac” follow the example:

```
TreeC <- gvisTreeMap(Regions,
"Region", "Parent",
"Fac")
plot(TreeC)
```

Exercise 9

Set “Fac” as your dependent variable, plot the tree map and see the difference.

Font size

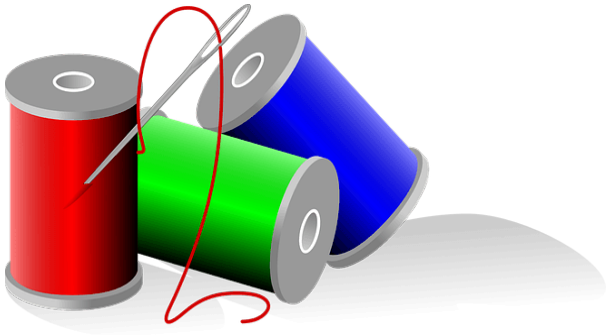
Obviously you can change the font size of your tree map simply with:

```
options=list(fontSize=10)
```

Exercise 10

Set the size of your font to 20 and plot your tree map. **HINT**: Use fontSize.

Hacking strings with stringr



This is first of the set of exercise on string manipulation with stringr

Answers to the exercises are available here.

If you obtained a different (correct) answer than those listed on the solutions page, please feel free to post your answer as a comment on that page.

Exercise 1
use a stringr function to merge this 3 strings .
x <- "I AM SAM. I AM SAM. SAM I AM"
y <- "THAT SAM-I-AM! THAT SAM-I-AM! I DO NOT LIKE THAT SAM-I-AM!"
z <- ""DO WOULD YOU LIKE GREEN EGGS AND HAM?"

Exercise 2

Now use a vector which contains x,y,z and NA and make it a single sentence using paste ,do the same by the same function you used for exercise1 .Can you spot the difference .

Exercise 3

Install the babynames dataset ,find the vector of length of the babynames using stringr functions. You may wonder nchar can do the same so why not use that ,try finding out the

difference and let me know in the comments.

Exercise 4

We often use substr to get part of the string ,in stringr world there exist a much powerful function which does almost the same thing . Create a string name with your name .
Use str_sub to get the last character and the last 5 characters .

Exercise 5

In mtcars dataset rownames, find all cars of the brand Merc .

Learn more about Text analysis in the online course Text Analytics/Text Mining Using R. In this course you will learn how create, analyse and finally visualize your text based data source. Having all the steps easily outlined will be a great reference source for future work.

Exercise 6

Use the same mtcars rownames ,find the total number of times “e” appears in that .

Exercise 7

Suppose you have a string like this
j <- "The_quick_brown_fox_jumps_over_the_lazy_dog"
split it in words using a stringr function

Exercise 8

On the same string I need the first word splitted but the rest intact ,help me to achieve that

Exercise 9

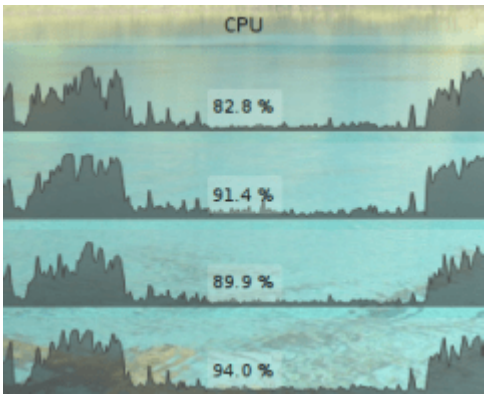
Now for on the same string J
a> I want the first “_” replaced by “–”
b> I want all the “_” replaced by “–”

Exercise 10

Many of the times ,you don't want NA to appear when you do some string manipulation but its sometimes necessary to replace NA as a character(rather than remove it) ,stringr provides a useful tool for that.
Now if I have a vector like this ,
na_string_vec <- c("The_quick_brown_fox_jumps_over_the_lazy_dog",NA)

How can you turn the NA into a character string .

Parallel Computing Exercises: Foreach and DoParallel (Part-2)



In general, foreach is a statement for iterating over items in a collection without using any explicit counter. In R, it is also a way to run code in parallel, which may be more convenient and readable that the sfLapply function (considered in the previous set of exercises of this series) or other apply-alike functions.
Apart from being able to run code in parallel, the R's foreach

has some other differences from the standard for loop. Specifically, the foreach statement:

- allows to iterate over several variables simultaneously,
- returns a value (a list, a vector, a matrix, or another object),
- is able to skip some iterations based on a condition (the last two properties make it similar to the list comprehension, which is present in Python and some other languages),
- has a special syntax that includes operators %do% (see an example in Exercise 1), %dopar%, and %:%.

The first six exercises in this set allow to train in performing basic operations with the foreach statement, and the last four ones show how to run it in parallel using multiple CPU cores on one machine. The task will be to parallelize identical operations on a set of files (the zipped data files can be downloaded here). It is assumed that your computer has two or more CPU cores.
The exercises require the packages foreach, doParallel, and parallel. The first two packages have to be installed, and the last one comes with the standard R distribution. The packages doParallel and parallel are necessary to run foreach in parallel.
For other parts of the series follow the tag parallel computing.
Answers to the exercises are available here.

Exercise 1
The foreach function (from the package of the same name) is typically used as a part of a special statement. In its simple form, the statement looks like this:

```
result <- foreach(i = 1:3) %do% sqrt(i)
```

The statement above consists of three parts:

- foreach(i = 1:3) – a call to the foreach function, with

an argument that includes an iteration variable (i) and a sequence to be iterated over (1:3),
- %do% – a special operator,
- sqrt(i): an R expression, which represents an operation to be performed over the iteration variable (this part of the statement is equivalent to the body of the loop).

The code iterates over the sequence, applies an operation defined in the expression to each element of the sequence, and stores the output in the result variable.
Note that if the expression extends over several lines it has to be enclosed in curly braces. The use of the iteration variable is not mandatory: if you just want to repeat the expression n times not passing anything to that expression you can use only a sequence of the length n as input to foreach.
In this exercise:

1. Run the code above, print the result object, and find to which class it belongs.
2. Use the foreach function to reverse the result. I.e. write a line of code that receives the result object as an input, and outputs the original sequence. Print the sequence.

Exercise 2
The foreach function allows for the use of several iteration variables simultaneously. They are passed to the function as arguments, and are separated by commas.
Run the foreach function with two iteration variables to get a sequence of their sums. The variables have to iterate over a vector of integers from 1 to 3, and a vector of 5 integers of value 10. Print the result.
(Tip: if you want to use an arithmetic operator to calculate the sum then the expression must be placed in parentheses or curly braces).
What is the length of the resulting object? How does the function deal with the vectors of different length?

Exercise 3

The package iterators provides several functions that can be used to create sequences for the foreach function. For example, the irnorm function creates an object that iterates over vectors of normally distributed random numbers. It is useful when you need to use random variables drawn from one distribution in an expression that is run in parallel.

In this exercise, use the foreach and irnorm functions to iterate over 3 vectors, each containing 5 random variables. Find the largest value in each vector, and print those largest values.

Before running the foreach function set the seed to 1234.

Exercise 4

By default the foreach function returns a list. But it can also return sequences of other types. This requires changing the value of the .combine parameter of the function. This exercise will train how to use this parameter.

As in the previous exercise, use the foreach and irnorm functions to iterate over 3 vectors, each containing 5 random variables. But now use an expression that returns all variables generated by irnorm. Pass the .combine parameter to the foreach function with value 'c'. Print the result, and find its class and length.

Then run the code again with the 'cbind' value assigned to the .combine parameter. Print the result, find its class and size.

Note that 'c' and 'cbind' are R functions from the base package. Other functions (including user-written ones) can be used as well to combine the outputs of the expression.

Exercise 5
The results of the expression placed after the %do% operator can be combined in different ways. Look at the documentation for the foreach function to find what value has to be assigned to the .combine parameter to sum the values produced by the expression in each iteration.
Run the code used in previous exercise with that value assigned to the .combine parameter, and print the result.
Before running the code set the seed to 1234.

Exercise 6
The sequence passed to the foreach function can be filtered so that the expression after %do% is applied only to a part of the sequence. This is done using a syntax like this:

```
result ‹- foreach(i = some_sequence) %:% when(i › 0) %do%
                                sqrt(i)
```

You can notice that the %:% operator and the when function, which contains a Boolean expression involving the iteration variable, are added to a standard foreach statement.
Modify the example above to get a vector of logs of all even integers in the range from 1 to 10. Print the result.

Exercise 7
Now let’s parallelize the execution of the foreach function. We’ll use it to read similarly named files, and perform identical calculations on data from each file.
As a first step, write a function to be run in parallel. The function takes an integer as input, and performs the following actions:

1. Create a string (character vector) with a file name by concatenating constant parts of the name (test_data_, .csv) with the integer (example of possible result when

1 is used as integer: test_data_1.csv).
2. Read the file with the obtained name from the current working directory into a data frame.
3. Calculate mean values for each column in the data frame.
4. Return a vector of those values.

Exercise 8
The second step is to create a backend for parallel execution:

1. Make a cluster for parallel execution using the makeCluster function from the parallel package; pass the size of the cluster (i.e. the number of CPU cores that you want to be used in computations) as an argument to this function .
2. Register the cluster with the registerDoParallel function from the doParallel package.

Note that by default the makeCluster function creates a PSOCK cluster, which is an enhanced version of the SOCK cluster implemented in the snow package. Accordingly, the PSOCK cluster is a pool of worker processes that exchange data with the master process via sockets. The makeCluster function can also create other types of clusters.

Exercise 9
The last step is to run the foreach function to read and analyze 10 test files (contained in this archive) using the function created in Exercise 7. Combine the outputs of that function using rbind.
Perform this task twice:

1. with %do% operator, which evaluates the expression sequentially, and
2. with %dopar% operator, which evaluates the expression in parallel.

In both cases, measure the execution time using the the system.time function. Print the result of the last run.
IMPORTANT: after completing parallel computations stop the

cluster (created in Exercise 8) using the stopCluster function from the parallel package.

Exercise 10
Modify the code written in the Exercise 7 and Exercise 9 to calculate the mean and the variance of values contained in the first column in each file. The resulting object must be a two-column matrix with the first column representing means, and the second column describing variances (the number of rows must be equal to the number of files).
Repeat the actions listed in Exercise 8 to prepare a cluster for parallel execution, then run the modified code in parallel.
Print the result.
Stop the cluster.

Data Visualization with googleVis exercises part 6

Geographical Charts

In part 6 of this series we are going to see some amazing geographical charts that googleVis provides.

Read the examples below to understand the logic of what we are going to do and then test yous skills with the exercise set we prepared for you. Lets begin!

Answers to the exercises are available here.

Package

As you already know, the first thing you have to do is install and load the googleVis package with:

```
install.packages("googleVis")
library(googleVis)
```

NOTE: The charts are created locally by your browser. In case they are not displayed at once press F5 to reload the page.

Geo Chart

It is quite simple to create a Geo Chart with googleVis. We will use the "Exports" dataset. First let's take a look at it with head(Exports). As you can see there are three variables ("Country", "Profit", "Online") which we are going to use later.
Look at the example below to create a simple geo chart:

```
Geo=gvisGeoChart(Exports )
plot(Geo)
```

Exercise 1

Create a list named "GeoC" and pass to it the "Exports" dataset as a geo chart. **HINT**: Use gvisGeoChart().

Exercise 2

Plot the the geo chart. **HINT**: Use plot().

Furthermore you can add much more information in your chart by using the locationvar and colorvar options to color the countries according to the their profit. Look at the example below.

```
Geo=gvisGeoChart(Exports,
locationvar="Country",
colorvar="Profit")
plot(Geo)
```

Exercise 3

Color the countries of your geo chart according to their profit and plot it. **HINT**: Use locationvar and colorvar.

Google Maps

It is quite simple to create a Google Map with googleVis. We will use the "Andrew" dataset. First let's take a look at it with head(Andrew) to see its variables. Look at the example below to create a simple google map:

```
GoogleMap <- gvisMap(Andrew)
plot(GoogleMap)
```

Exercise 4

Create a list named "GoogleMap" and pass to it the "Andrew" dataset as a google map. **HINT**: Use gvisMap().

Learn more about using GoogleVis in the online course Mastering in Visualization with R programming. In this course you will learn how to:

- Work extensively with the GoogleVis package and its functionality
- Learn what visualizations exist for your specific use case
- And much more

Exercise 5

Plot the the google map. **HINT**: Use plot().

As you can see there ane no data points on it as we did not select something yet. We have to select the latitude and longtitude variables for the dataset like the example below.

```
GoogleMap <- gvisMap(Andrew,"LatLong" )
```

Exercise 6

Display the map by addind the “LatLong” variable to your list and plot it.

Exercise 7

Display the “Tip” variable on your google map just like you displayed the “LatLong” and plot it.

There are some useful options that gvisMap() provides to you that can enhance your map. Check the example below.

```
options=list(showTip=TRUE,
showLine=TRUE,
mapType='terrain',
useMapTypeControl=TRUE)
```

Exercise 8

Deactivate the Tip information from your map, plot the map and then enable it again. **HINT**: Use showTip.

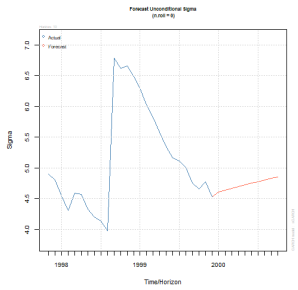
Exercise 9

Enable useMapTypeControl and plot the map.

Exercise 10

Set the mapType by default to “terrain” and plot the map.

Volatility modelling in R exercises (Part-3)



This is the third part of the series on volatility modelling. For other parts of the series follow the tag volatility.

In this exercise set we will use GARCH models to forecast volatility.

Answers to the exercises are available here.

Exercise 1
Load the rugarch and the FinTS packages. Next, load the m.ibmspln dataset from the FinTS package. This dataset contains monthly excess returns of the S&P500 index from Jan-1926 to Dec-1999 (Ruey Tsay (2005) Analysis of Financial Time Series, 2nd ed. ,Wiley, chapter 3).
Also, load the forecast package which we will use for auto-correlation graphs.

Exercise 2
Excess S&P500 returns are defined as a regular zoo variable. Convert this to a time series variable with correct dates.

Exercise 3
Plot the excess S&P500 returns along with its ACF and PACF graphs and comment on the apparent correlation.

Exercise 4
Plot the squared excess S&P500 returns along with its ACF and PACF graphs and comment on the apparent correlation.

learn how to:

- Avoid model over-fitting using cross-validation for optimal parameter selection
- Explore maximum margin methods such as best penalty of error term support vector machines with linear and non-linear kernels.
- And much more

Exercise 5
Using the results from exercise-3, estimate a suitable ARMA model for excess returns assuming normal errors.

Exercise 6
Using the results from exercise-4, estimate a suitable ARMA model for excess returns without assuming normal errors.

Exercise 7
Using the results from exercises 5 and 6, estimate a more parsimonious model that has better fit.

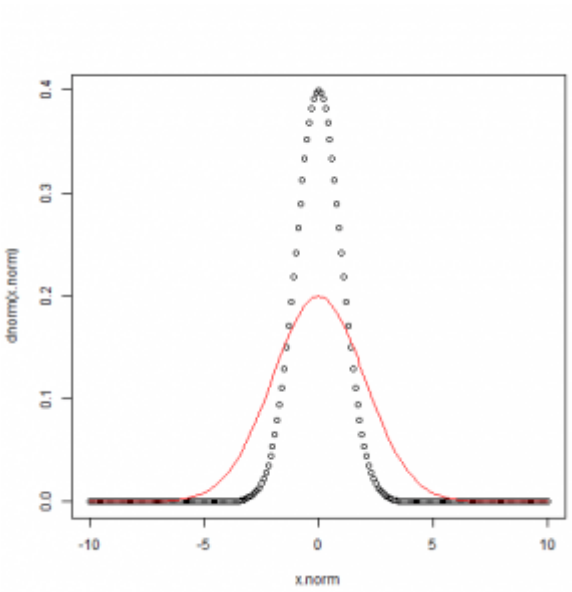
Exercise 8
Generate 10 steps ahead forecast for the model from exercise-7

Exercise 9
Plot the excess returns forecast.

Exercise 10
Plot the volatility forecast.

Hacking statistics or: How I

Learned to Stop Worrying About Calculus and Love Stats Exercises (Part-2)



Statistics are often taught in school by and for people who like Mathematics. As a consequence, in those class emphasis is put on leaning equations, solving calculus problems and creating mathematics models instead of building an intuition for probabilistic problems. But, if you read this, you know a bit of R programming and have access to a computer that is really good at computing stuff! So let's learn how we can tackle useful statistic problems by writing simple R query and how to think in probabilistic terms.

In the last exercise set we've seen that random variable can be described by mathematical functions called probability density and that when we know which one describe a particular random process we can use it to compute the probability of realization of a given event. We have also seen how to use an histogram and an ECDF plot to identify which function express the random variable. Today, we will see which mathematical properties of those function we can compute to help us find the probability density who fit a sample. Those properties are called statistics and our job today is to estimate the real value of those properties by using a small sample of data.

Answers to the exercises are available here.

Exercise 1

The most commonly used statistics is the mean, which is the center of mass of the distribution, i.e. the point on the x axis where the weighted relative position of each observation sum to zero. For example, draw the density of a standard normal distribution and add to the plot a vertical line to indicate the mean of this distribution. Then, draw another plot, but this time of an exponential distribution with a rate of 1 and his mean.

From the density plot of the standard normal distribution we can see how the mean represent the center of mass of the distribution: the normal distribution is symmetric, so the mean is in the center of the plot of the function. The exponential function is not symmetric, in this case the mean is the point where all the points with a small y value, at the right of the mean on the plot, counterbalance the few points with a high y value at the left of the mean. Since the value of the mean is at the center of the distribution, we often use the mean to represent a typical value of a probability distribution. The mean also give us the ability to put a number on the location of a probability distribution on the axis.

Exercise 2

In practice, we don’t have access to the probability density function of a random variable and can’t compute directly the mean of the distribution. We must estimate it using a sample of observations of that random variable. Since it’s random, all samples will be different and our estimations of the mean, will all be different.

Generate 500 points from an exponential distribution with a rate of 0.5. Draw the histogram of the sample and compute the sample mean of this distribution. Then write a function that repeat this process for n iterations, store the sample mean in a vector and return this vector. Use this function to compute 10,000 sample means, plot the histogram of the sample means and compute the mean of those estimations.

Exercise 3

From the histogram of the sample mean, we can see that the estimations follow a normal distribution centered around the real value of the mean. We can use this fact to compute the interval which have a certain probability of containing the real value of the mean. This interval is called the confidence interval of the estimate and the probability that this interval contain the real mean of the distribution is called the confidence level. In the next exercise set, we will see methods to compute this interval directly from a sample without knowing the probability density function of the random variable.

Use the quantile() function to compute the 2.5 and 97.5 percentile from the sample of estimations of the mean, then use the t.test() function to compute the confidence interval with a level of 95% of the original distribution and compare those values.

Exercise 4

Load this dataset and use the t.test() function to compute the confidence interval of the mean for both variables with a level of 95%. Does those random variables seems to follow distributions who have the same means?

We see that the confidence intervals doesn’t overlap. This is an indication that the real value of the mean of the first

variable is not in the same interval as the distribution mean of the second variable. As a consequence, we can safely suppose than both mean are different and that they don’t have the same probability distribution.

Exercise 5
Another useful statistics is the variance. This statistic is an indication of how the data are spread around the mean. So if two distributions have the same mean, the one with the smallest variance has the most homogeneous value, while the one with the highest variance has more small and high value far from the mean. A related statistics is the standard deviation, which is defined as the square root of the variance.

Draw the density of a standard normal distribution and of a normal distribution of mean equal to zero and with a standard deviation of 5 to see the effect of a change of variance on a density.

Exercise 6
In the case of the variance, we cannot directly compute the confidence interval without making assumption on the type of distribution the sample come from or use some fancy method we will introduce in the next exercise set. Luckly for us we can use thevar.test() function to verify if the variances are equal. Use this function on the dataset of exercise 4 three time, once with the alternative parameter set to “two.sided”, then to “less” and finally to “greater”. What is the signification of the three test?

Exercise 7
If the mean is a good representation of the typical value of a random variable defined by a density, this statistics can be skew by outliers. When a sample has outlier a better statistics to use is the median, which is the value that separate the range of observations that can be generated by a random variable in two equal parts.

Generate 200 points from a log-normal distribution with a parameter meanlog = 0 and sdlog = 0.5. Then plot the histogram of those points and represent the mean and the median of this sample by two vertical lines.

Exercise 8

The median is a special case of a more general statistics called quantile, which are cutpoints dividing the domain of a probability density function into sub-interval containing the same amount of observations. So the 2-quantile is the median, since this statistics separate the domain of a probability distribution in two sub-interval containing 50% of the observations. Other quantile statistics often used are the 4-quantile, called quartile, which are the values on the domain of a probability distribution that separates it in four sub-interval containing 25% of the observations and the 100-quantile, called percentille, which are the values that separate this domain in 100 part containing 1% of the observations.

Compute the median, the quantile and the 5 and 95% percentile on the variables of the dataset of exercise 4. Then compute the interquartile range which is the difference between the 25% and the 75% quartile. Does those statistics suggest that the two samples have the same distribution?

Exercise 9

Another statistics that can be used to differentiate two probability distribution is the skewness. As his name imply, the skewness is a measure of how much there is an imbalance between the observations at the right of the mean and at the left of the mean. A negative skewness indicate that the distribution is skew to the left, a positive value indicate that the distribution is skew to the right and a skewness of zero tell us that the distribution is perfectly symmetric.

Load the moment package and use the skewness() function to compute the skewness of three samples you must create:

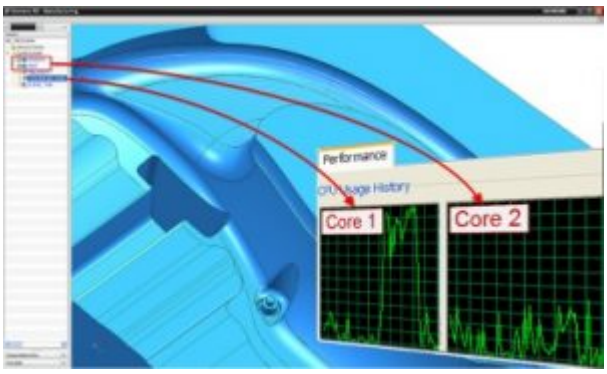
- 150 points sample from a standard normal distribution
- 1000 points sample from a standard normal distribution
- 200 points sample from a exponential distribution with a rate of 5

Exercise 10
The last statistic we will use today is the kurtosis, which describe the general shape of the probability distribution. When the kurtosis is greater than zero, the probability distribution has heavy tail and a pointy shape. Both of those characteristics are proportional to the magnitude of the kurtosis. If the kurtosis is less than zero, the distribution has a more regular shape with light tails. When this statistic has a value of zero, the distribution's shape look a lot like the normal distribution.

Use the kurtosis() function to compute the kurtosis of those samples:

- 500 points sample from a standard normal distribution
- 500 points sample from a exponential distribution with a rate of 5
- 500 points sample from uniform distribution

Parallel Computing Exercises: Snowfall (Part-1)



R has a lot of tools to speed up computations making use of multiple CPU cores either on one computer, or on multiple machines. This series of exercises aims to introduce the basic techniques for implementing parallel computations using multiple CPU cores on one machine.

The initial step in preparation for parallelizing computations is to decide whether the task can and should be run in parallel. Some tasks involve sequential computation, where operations in one round depend on the results of the previous round. Such computations cannot be parallelized. The next question is whether it is worth to use parallel computations. On the one hand, running tasks in parallel may reduce computer time spent on calculations. On the other hand, it requires additional time to write the code that can be run in parallel, and check whether it yields correct results.

The code that implements parallel computations basically makes three things:

- splits the task into pieces,
- runs them in parallel, and
- combines the results.

This set of exercises allows to train in using the snowfall package to perform parallel computations. The set is based on the example of parallelizing the k-means algorithm, which splits data into clusters (i.e. splits data points into groups based on their similarity). The standard k-means algorithm is sensitive to the choice of initial points. So it is advisable to run the algorithm multiple times, with different initial points to get the best result. It is assumed that your

computer has two or more CPU cores.
The data for the exercises can be downloaded here.
For other parts of the series follow the tag parallel computing.
Answers to the exercises are available here.

Exercise 1
Use the detectCores function from the parallel package to find the number of physical CPU cores on your computer. Then change the arguments of the function to find the number of logical CPU cores.

Exercise 2
Load the data set, and assign it to the df variable.

Exercise 3
Use the system.time function to measure the time spent on execution of the command fit_30 <- kmeans(df, centers = 3, nstart = 30), which finds three clusters in the data.
Note that this command runs the kmeans function 30 times sequentially with different (randomly chosen) initial points, and then selects the 'best' way of clustering (the one that minimizes the squared sum of distances between each data point and its cluster center).

Exercise 4
Now we'll try to paralellize the runs of kmeans. The first

step is to write the code that performs a single run of the kmeans function. The code has to do the following:

1. Randomly choose three rows in the data set (this can be done using the sample function).
2. Subset the data set keeping only the chosen rows (they will be used as initial points in the k-means algorithm).
3. Transform the obtained subset into a matrix.
4. Run the kmeans function using the original data set, the obtained matrix (as the centers argument), and without the nstart argument.

Exercise 5
The second step is to wrap the code written in the previous exercise into a function. It should take one argument, which is not used (see explanation on the solutions page), and should return the output of the kmeans function.
Such functions are often labelled as wrapper, but they may have any possible name.

Exercise 6
Let’s prepare for parallel execution of the function:

1. Initialize a cluster for parallel computations using the sfInit function from the snowfall package. Set the parallel argument equal to TRUE. If your machine has two logical CPU’s assign two to the cpus argument; if the number of CPU’s exceeds two set this argument equal to the number of logical CPU’s on your machine minus one.
2. Make the data set available for parallel processes with the sfExport function.
3. Prepare the random number generation for parallel execution using the sfClusterSetupRNG. Set the seed argument equal to 1234.

(Note that kmeans is a function from the base R packages. If your want to run in parallel a function from a downloaded

package, you have also to make it available for parallel execution with the sfLibrary function).

Exercise 7
Use the sfLapply function from the snowfall package to run the wrapper function (written in Exercise 5) 30 times in parallel, and store the output of sfLapply in the result variable. Apply also the system.time function to measure the time spent on execution of sfLapply.
Note that sfLapply is a parallel version of lapply function. It takes two main arguments: (1) a vector or a list (in this case it should be a numeric vector of length 30), and (2) the function to be applied to each element of the vector or list.

Exercise 8
Stop the cluster for parallel execution with the sfStop function from the snowfall package.

Exercise 9
Explore the output of sfLapply (the result object):

1. Find out to what class it belongs.
2. Print its length.
3. Print the structure of its first element.
4. Find the value of the tot.withinss sub-element in the first element (it represents the total squared sum of distances between data points and their cluster centers in a given solution to the clustering problem). Print that value.

Exercise 10
Find an element of the result object with the lowest tot.withinss value (there may be multiple such elements), and assign it to the best_result variable.
Compare the tot.withinss value of that variable with the corresponding value of the fit_30 variable, which was obtained in Exercise 3.