# Minitask1

Florencia Luque and Seyed Amirhossein Mosaddad

2024-10-10

## Part 1 - Apply Family

The apply Family is a way to apply a function to different types of data like a row, vector, matrix, etc. This functions have been replace or you should replace it with others libraries that do the same with an optimal result. This function is an alternative to a loop when you know waht you need to do to the data.

The family is created by these functions; apply(), lapply() , sapply(), vapply(), mapply() and tapply() functions.

### apply

This function can be applied to arrays in different dimensions like a matrix. You will need the array(matrix), Margin 1 or 2 if you want to apply the function by row (1) and over columns (2). You can also apply to both by using "Margin=c(1,2)" and obviously the "FUN" method, which is the function that will be applied. For example the sum or the mean. Example:

```
mat = matrix(data = rnorm(20,20,4),nrow = 4,ncol = 5,byrow = TRUE)
print(mat)
```

```
##          [,1]     [,2]     [,3]     [,4]     [,5]
## [1,] 15.25327 20.72747 20.60747 15.81200 12.76572
## [2,] 16.67942 17.26058 22.34359 18.33938 12.78723
## [3,] 22.21938 21.97416 20.97191 27.29359 16.21198
## [4,] 20.93722 16.46432 22.55729 18.23560 19.29875
```

```
sum_by_column = apply(mat,MARGIN = 2,FUN = sum)

print(sum_by_column)
```

```
## [1] 75.08928 76.42653 86.48025 79.68056 61.06369
```

### tapply

This function is very similar to apply but you can apply the function by groups. You can use this function to a data frame or a matrix and you will need the data, the index that will be the factor that you want to group by, and the FUN method. We can also have more optional arguments. We can transform the data to aggregate a factor variable and create an example.

```r
data = data.frame(mat)
data$grupo = c("A","B","B","A")
colnames(data)[1] = "age"
colnames(data)[2] = "bmi"
data$grupo = as.factor(data$grupo)

tapply(data$age,data$grupo,mean)
```

```
##        A        B
## 18.09524 19.44940
```

### lapply

This function is applied over a list or a vector and returns a list of the same length of the original input. This is a way to apply a function to every element in a list. The function will need a list or a vector and the function that you want to apply. This means that if you give it a vector it will apply the function to each element and if you give it a list it will apply to the elements of the list.

```r
lista = list(tr = c(1:6),mn = c(5:8))
lista
```

```
## $tr
## [1] 1 2 3 4 5 6
##
## $mn
## [1] 5 6 7 8
```

```r
lapply(lista, median)
```

```
## $tr
## [1] 3.5
##
## $mn
## [1] 6.5
```

```r
vec = sample(1:40,5)
vec
```

```
## [1] 28  7 13 39 12
```

```r
lapply(vec, FUN = function(x) 0.5*x)
```

```
## [[1]]
## [1] 14
##
## [[2]]
## [1] 3.5
##
## [[3]]
## [1] 6.5
```

```
##
## [[4]]
## [1] 19.5
##
## [[5]]
## [1] 6
```

### sapply

This function can be applied to a vector or list and will return a vector or a list respectively. This function is very similar to lapply but it tries to simplify it. It is a wrapper of the lapply it can return not only a list but also a vector, matrix or an array. Let's do the same exercise to see the difference between both functions.

```
lista = list(tr = c(1:6),mn = c(5:8))
lista
```

```
## $tr
## [1] 1 2 3 4 5 6
##
## $mn
## [1] 5 6 7 8
```

```
sapply(lista, median)
```

```
##  tr  mn
## 3.5 6.5
```

```
vec = sample(1:40,5)
vec
```

```
## [1]  1 26 32 10 11
```

```
sapply(vec, FUN = function(x) 0.5*x)
```

```
## [1]  0.5 13.0 16.0  5.0  5.5
```

This function gives you the same type that you give it in the input.

### mapply

This is the multivariate version of sapply. Applying a function to multiple arguments.

```
mapply(sum, 1:5, 6:10)
```

```
## [1]  7  9 11 13 15
```

### vapply

This is a more strict version of sapply where you need to specify the type of return that you would like.

```r
my_list <- list(a = 1:5, b = 6:10)
vapply(my_list, sum,numeric(1))
```

```
##  a  b
## 15 40
```

# Part 2

## MLE for gamma distribution

### Differentiating with Respect to $\theta$

Given a gamma distribution with parameters $k$ (shape) and $\theta$ (scale), the probability density function is:

$$f(x \mid k, \theta) = \frac{1}{\Gamma(k)\theta^k} x^{k-1} e^{-x/\theta}$$

### Log-Likelihood Function

The likelihood function for a sample $(x_1, x_2, \ldots, x_n)$ is given by:

$$L(x_1, \ldots, x_n \mid k, \theta) = \prod_{i=1}^{n} f(x_i \mid k, \theta) = \left(\frac{1}{\Gamma(k)\theta^k}\right)^n \prod_{i=1}^{n} x_i^{k-1} e^{-x_i/\theta}$$

Taking the logarithm of the likelihood function, we get the log-likelihood:

$$\log L(k, \theta) = -n \log \Gamma(k) - nk \log \theta + (k-1) \sum_{i=1}^{n} \log x_i - \frac{1}{\theta} \sum_{i=1}^{n} x_i$$

To find the maximum likelihood estimate of $\theta$, we take the partial derivative of the log-likelihood with respect to $\theta$ and set it to zero:

$$\frac{\partial \log L(k, \theta)}{\partial \theta} = -\frac{nk}{\theta} + \frac{\sum_{i=1}^{n} x_i}{\theta^2}$$

Setting this equal to zero and solving for $\theta$:

$$-\frac{nk}{\theta} + \frac{\sum_{i=1}^{n} x_i}{\theta^2} = 0$$

This simplifies to:

$$\theta = \frac{\sum_{i=1}^{n} x_i}{nk}$$

**Final form for $\hat{\theta}$**

Thus, the maximum likelihood estimate of the scale parameter $\theta$ is:

$$\hat{\theta} = \frac{\bar{x}}{k}$$

where $\bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i$.

**Differentiating with Respect to $k$**

The log-likelihood function for a sample $x_1, x_2, \ldots, x_n$ from a Gamma distribution with shape parameter $k$ and scale parameter $\theta$ is:

$$\log L(k, \theta) = -n \log \Gamma(k) - nk \log \theta + (k-1) \sum_{i=1}^{n} \log x_i - \frac{1}{\theta} \sum_{i=1}^{n} x_i.$$

1. **First Term:** $-n \log \Gamma(k)$

$$\frac{\partial}{\partial k}[-n \log \Gamma(k)] = -n \frac{\Gamma'(k)}{\Gamma(k)},$$

   where $\frac{\Gamma'(k)}{\Gamma(k)}$ is the digamma function $\psi(k)$.

2. **Second Term:** $-nk \log \theta$

$$\frac{\partial}{\partial k}[-nk \log \theta] = -n \log \theta.$$

3. **Third Term:** $(k-1)\sum_{i=1}^{n} \log x_i$

$$\frac{\partial}{\partial k}\left[(k-1) \sum_{i=1}^{n} \log x_i\right] = \sum_{i=1}^{n} \log x_i.$$

4. **Fourth Term:** $-\frac{1}{\theta}\sum_{i=1}^{n} x_i$

   This term does not depend on $k$, so its derivative with respect to $k$ is zero.

**Combining the Derivatives**

Putting these together, the derivative of the log-likelihood with respect to $k$ is:

$$\frac{\partial \log L(k, \theta)}{\partial k} = -n\psi(k) - n \log \theta + \sum_{i=1}^{n} \log x_i.$$

As we can see, we are not able to solve for $k$ analytically as it does not have a closed-form solution. Hence, we need to use numerical methods to solve for $k$.

```r
# we need to install the required library
if (!requireNamespace("stats", quietly = TRUE)) {
  install.packages("stats")
}
```

Most optimisation algorithms are designed and implemented in a way to compute the minimisation of the function rather than maximise it. But here we would like to estimate the maximum likelihood by maximising the log-likelihood function. So we need to convert our function into the negative log-likelihood.

```
gamma_negativeLL = function(params, data) {
  k = params[1]
  theta = params[2]
  n = length(data)
  negativeLL = n*(log(gamma(k)) + (k*log(theta))) - (k - 1) * sum(log(data)) + (sum(data) / theta)
  return (negativeLL)
}
```

Given that:

$$X \sim \text{Gamma}(k, \theta)$$

- The mean of the gamma distribution is:

$$\mu = k\theta$$

- The variance of the gamma distribution is:

$$\sigma^2 = k\theta^2$$

From these, we can derive the following estimators using methods of moments:

- The sample mean:

$$\bar{X} = k\theta$$

Therefore,

$$\theta = \frac{S^2}{\bar{X}}$$

- The sample variance:

$$S^2 = k\theta^2$$

Thus, the shape parameter $k$ can be estimated as:

$$k = \frac{\bar{X}^2}{S^2}$$

```r
library(stats4)
mleGamma = function(data) {
  xBar = mean(data)
  xVar = var(data)
  initK = (xBar^2)/xVar
  initTheta = xVar/xBar

  # since we want to maximise a multivariate function (function with more than one parameter),
  # we use the function mle, which is used for multidimensional optimisation
  mleResult = mle(
    minuslogl = function(k, theta) gamma_negativeLL(c(k, theta), data),
    start = list(k = initK, theta = initTheta), #starting values
    method = "L-BFGS-B", # optimisation method
    lower =  c(1e-6, 1e-6), # lower bounds to avoid invalid params
    upper = c(Inf, Inf) # upper bound
  )
  return (coef(mleResult))
}
```

**side note:**

We specifically use the "L-BFGS-B" methods for optimisation as this method is useful for solving problems involving bound constraints on the parameters. The gamma distribution parameters $\theta$ and $k$ must be strictly positive. Hence by using this methd we can set a lower bound to ensure they remain withtin a valid range.

Now we set a seed to be able to reproduce the same result and then we use the "*rgamma*" function to generate data with gamma distribution.

```r
set.seed(43)
sampleData = rgamma(1000, shape = 2, scale = 3)
```

Now we run our MLE estimator to see how close they are to our true parameters.

```r
results = mleGamma(sampleData)
cat("Estimated shape (k):", results[1], "\n")
```

```
## Estimated shape (k): 2.067722
```

```r
cat("Estimated scale (theta):", results[2], "\n")
```

```
## Estimated scale (theta): 2.916641
```

## MaxLik in R

MaxLik is an R package designed and implemented to calculate Maximum Likelihood Estimation (MLE) for various statistical models. It offers useful tools and built-in functions to facilitate parameter estimation and analysis for different distributions.

This framework provides functionalities to easily compute the estimation of complex models without havin to manually derive estimation equations. It also offers features to check and interpet the results.

A summary of more important properties and features is listed below:

1. This framework has a lot of different optimisation methods that allow users to easily handle their statistical models. In addition, users can define their own custom log-likelihood functions and analyse more complex models if it is not pre-defined in the framework.

2. There are many optimisation algorithms available to choose from. some of which are:

   - Newton-Raphson: A second order method to find the optimum.
   - Berndt-Hall-Hall-Hausman (BHHH): A variant of Newton-Raphson that utilises the outer product of the gradient.
   - BFGS (Broyden-Fletcher-Goldfarb-Shanno): A quasi-Newton method that approximates the Hessian matrix.
   - Simplex and Nelder-Mead: Non-gradient methods for optimizing non-differentiable functions.
   - Simulated Annealing: A probabilistic method for global optimization.

3. In addition, we are able to calculate standard errors using different methods:

   - Hessian Matrix: we can compute standard errors by calculating the Hessian Matrix, which is usually derived from the second derivative of the likelihood function.
   - Outer Product of Gradients (OPG): By utilising the outer product of the gradient vector.
   - Sandwich Estimator: It accounts for model misspecification.

4. With MaxLik, we can impose linear or non-linear constraints on the parameters space and optimise our model that satisfies the parameter conditions.

5. Moreover, we can check the quality of our estimated model with MaxLik by using a range of post-estimation tools:

   - Likelihood Ratio Tests: To compare models and evaluate if additional parameters improve the model fit.
   - Confidence Intervals: Computed for estimated parameters to provide estimation uncertainty.
   - Gradient and Hessian Checking: To check the accuracy of the numerical optimization and see if we have issues like non-convergence or improper model specification.

6. MaxLik offers a very user-friendly and intuitive interface. It simplifies MLE even for users without expertise in optimisation. The model need to be specified in normal R syntax and MaxLik can do the rest.

7. There are some advanced features availabe as well for more complex models and optimisations:

   - Multi-parameter Estimation: Supports estimation of multiple parameters simultaneously, such as the Gamma distribution.
   - Support for Non-standard Likelihoods: handling likelihood functions that do not follow the standard forms. It allows users to implement customised models.
   - Parallel Computing: to speed up optimisation. More effective for large-scale problems and models with high computational need

8. MaxLik also integrates with other R packages, making it easier to be utilised for larger workflows. For example, it can be integrated with data visualisation packages and other statistical frameworks.

**Applications**

MaxLik can be applied in any fiels that requires statistical modeling and optimisation. some of which are:

1. Econometrics: For estimating economic models like logistic regression, probit models, and time series analysis.

2. Biostatistics: In survival analysis and other medical research models where likelihood-based estimation is essential.

3. Machine Learning: For optimizing loss functions in supervised learning models.

### Limitations

MaxLik also has its limitations. For example:

1. Numerical Sensitivity: The results of MLE can be sensitive to the choice of starting values and the optimization method. Poor choices can cause convergence issues or incorrect estimates.

2. Computational intensity: For large datasets or complex models, MaxLik can be computationally intensive and slow, particularly when using numerical approximations for gradients and Hessians.

### Typical Workflow

Here is an example of a typical workflow in MaxLik:

1. We first define the log-likelihood functions with data and a set of parameters being the inputs. Here we have a simple log-likelihood function for linear regression model:

```
log_likelihood = function(params, data) {
    # Extract parameters
    param1 = params[1]
    param2 = params[2]
    # ...

    # Calculate log-likelihood based on your model
    ll = sum(log_probability_density_function(...))

    return(ll)
}
```

2. The main function in the MaxLik library is maxLik(), which performs maximum likelihood estimation.

```
result = maxLik(logLik = log_likelihood,
                grad = gradient,  # Optional
                hess = hessian,   # Optional
                start = start_values,
                method = "BFGS",  # or another method
                data = data,
                nproc = 4)        # For parallel processing
```

3. Then we extract the results for desired estimated parameters.

```
# View summary of results
summary(result)

# Extract coefficients
coefficients <- coef(result)
```

```r
# Get standard errors
std_errors <- sqrt(diag(vcov(result)))

# Calculate confidence intervals
conf_intervals <- confint(result)
```

4. Finally, we can perform post-estimation analysis such as hypothesis tests, model fit, and confidence intervals.

```r
# Create tables of results
results_table = data.frame(
    Estimate = coefficients,
    Std.Error = std_errors,
    CI_lower = conf_intervals[,1],
    CI_upper = conf_intervals[,2]
)

# Plot results or diagnostics
plot_results(result)
```

### MaxLik for our gamma-distributed data

Now, we use this package to work out the MLE estimates for our gamma-dstributed data.

```r
# Check if maxLik is installed, if not, install it
if (!requireNamespace("maxLik", quietly = TRUE)) {
  install.packages("maxLik")
}

# This function remains the same, but we'll negate it for MaxLik
gamma_negativeLL = function(params, data) {
  k = params[1]
  theta = params[2]
  n = length(data)
  negativeLL = n*(log(gamma(k)) + (k*log(theta))) - (k - 1) * sum(log(data)) + (sum(data) / theta)
  return (negativeLL)
}

mleGamma = function(data) {
  xBar = mean(data)
  xVar = var(data)
  initK = (xBar^2)/xVar
  initTheta = xVar/xBar

  # MaxLik maximizes, so we need to negate our negative log-likelihood function
  mleResult = maxLik::maxLik(
    logLik = function(params) -gamma_negativeLL(params, data),
    start = c(k = initK, theta = initTheta),
    method = "BFGS"  # BFGS is similar to L-BFGS-B but doesn't require bounds
  )
  return (coef(mleResult))
```

```
}

# Generate sample data
set.seed(43)
sampleData = rgamma(1000, shape = 2, scale = 3)

# Estimate parameters
MaxLikResults = mleGamma(sampleData)
cat("Estimated shape (k):", MaxLikResults[1], "\n")
```

```
## Estimated shape (k): 2.067722
```

```
cat("Estimated scale (theta):", MaxLikResults[2], "\n")
```

```
## Estimated scale (theta): 2.916643
```

### Comparison

Now, we conduct the comparison between these two methods:

```
# Print results
cat("True parameters:\n")
```

```
## True parameters:
```

```
cat("Shape (k):", 2, "\n")
```

```
## Shape (k): 2
```

```
cat("Scale (theta):", 3, "\n\n")
```

```
## Scale (theta): 3
```

```
cat("Estimated parameters (maxLik):\n")
```

```
## Estimated parameters (maxLik):
```

```
cat("Shape (k):", MaxLikResults[1], "\n")
```

```
## Shape (k): 2.067722
```

```
cat("Scale (theta):", MaxLikResults[2], "\n\n")
```

```
## Scale (theta): 2.916643
```

```r
cat("Estimated parameters (stats4::mle):\n")
```

```
## Estimated parameters (stats4::mle):
```

```r
cat("Shape (k):", results[1], "\n")
```

```
## Shape (k): 2.067722
```

```r
cat("Scale (theta):", results[2], "\n\n")
```

```
## Scale (theta): 2.916641
```

```r
# Calculate percentage differences
pct_diff_k = (MaxLikResults[1] - results[1]) / results[1] * 100
pct_diff_theta = (MaxLikResults[2] - results[2]) / results[2] * 100

cat("Percentage differences (maxLik vs stats4::mle):\n")
```

```
## Percentage differences (maxLik vs stats4::mle):
```

```r
cat("Shape (k):", round(pct_diff_k, 4), "%\n")
```

```
## Shape (k): 0 %
```

```r
cat("Scale (theta):", round(pct_diff_theta, 4), "%\n")
```

```
## Scale (theta): 1e-04 %
```

As we can see, we get very similar results for ML estimations of $k$ and $\theta$ from both optimisation frameworks.