GuessCompx: an R package to Empirically Estimate Algorithm Complexity

by Marc Agenis-Nevers, Neeraj Dhanraj Bokde, Mayur Shende

Abstract This article introduces `GuessCompx`, an R package that makes an empirical guess on the time and memory complexities of an algorithm or a function. It will test multiple, increasing-sizes samples of the user's data and try to fit one of seven complexity functions: *O(N), O(N^2), O(log(N))*, etc. Based on a best fit procedure using LOO-MSE (leave one out-mean square error), it also predicts the full computation time and memory usage on the whole dataset. It relies on the base R functions `system.time` and `memory.size`, the latter being only suitable for Windows users. Together with this result, a plot and a significance test are also returned. Complexity is assessed with regard to the user's actual dataset through its size (and no other parameter). This article provide several examples showing some use cases (distance function, time series, custom function) and how to best tune the parameters. The subject of empirical computational complexity has been relatively little studied in computer sciences, and such a package provides a convenient and simple procedure to estimate it, thus preventing the user to run any computation for an unknown amount of time. Empirical fit does not guarantee to find the true complexity function but approaches it in an acceptable way. The package does not require to have the code of the target function.

Introduction

Complexity of an algorithm is a measure of amount of resources the algorithm needs to run to completion, for a given input (He and Yao, 2001)(Jensen, 2003). The resources an algorithm needs may be space or time, which represent two types of complexities (Woeginger, 2004). The time complexity is the total amount of time required by an algorithm to complete its execution. The running time of an algorithm may differ for the same input sizes, depending on the type of data (Valiant, 1979). For example, in bubble sort algorithm, if the input array is already sorted then the running time of the algorithm will be lower. This is called the best case complexity. Similarly, if the input list is in reverse order the algorithm will take maximum time, this is called worst case complexity. If the input list is neither sorted nor reversed, then the running time is less than worst case but more than best case: this is the average case complexity. Since an algorithm can't require more time than the worst case scenario, this scenario is set to be the reference complexity (Min, 2010)(Yang et al., 2011). Since this function is generally difficult to compute exactly, and the running time for small inputs is usually not consequential, one commonly focuses on the behavior of the complexity when the input size increases, that is, the asymptotic behavior of the complexity. Therefore, the time complexity is commonly expressed using big `O` notation, typically *O(N), O(NlogN), O(N^2), O(2^N)*, etc., where `N` is the input size in units of bits needed to represent the input (Chivers and Sleightholme, 2015).

A mathematical problem can be solved in different ways. Each of these ways might implement a different algorithm. These algorithms do not need the same amount of resources and can have different complexities. The algorithm that uses less memory and completes in less time is, by definition, more efficient. By studying the complexities of different algorithms, we can determine the most efficient algorithm for a given input.

In the specific field of data science, we witness the exponential growth of datasets size and required computational power. Some reviews like (Qiu et al., 2016) that gives an overview of the statistical methods for big data, mentions the words "computational complexity" or "computational efficiency" no less than 10 times. However, while designing new algorithms, their combinations, or even simply data treatments or data streams, the engineers have a constant need to check the computational complexity of their code. A great amount of time can be lost in this process, while waiting for a new computation to complete without knowing the exact time required: one minute, one hour, several weeks? Is it even worth waiting? Having a way to estimate the execution time of a new piece of code, before running it at full scale, could be a critical time-saving tool.

According to (Sharma et al., 2018), "*it takes significant amount of effort to judge the complexity of an algorithm. Various manual methods are used till now to get calculate the time complexities such as Master Method, using control flow graph, but all of them remain tedious to work with and owing to their manual nature, have limitations and are prone to error*". Recently, manual and exact methods have started to be challenged by so-called empirical methods, that try to give an estimate of the complexity by observing the code being run several times. Goldsmith et al. (2007) proposed a tool in C/C++ language named Trend Profiler (`trend-prof`) to construct models of empirical computational complexity that estimate how many times each basic block in a program runs; statistical modeling is applied, in the form of

linear, polynomial or power terms, the models for the different blocks of code are eventually combined to predict the final execution time or to report performance problems to the developer. The authors don't directly measure CPU-time and justify this choice by the desire to avoid the replicability problem. Modeling code-blocks is more robust but has a major drawback: it needs to access the source code of the target function and to ask some manual-input features like the number of nodes, characters, lines, etc. More recently, Sharma et al. (2018) introduced a concept of predicting time complexity using gradient boosted trees in a supervised manner in C++ language and Python. The authors reached exceptional prediction capability by treating the problem not as a regression anymore but as a classification into 7 complexity families. The algorithm has to be executed in a controlled environment with a high number of replications, to retrieve 6 features linked with execution time and hardware information (clock time, number of processors, etc.), but the technique also gave good results using only two of them: the execution time and processor speed. Unfortunately the code and the models are not directly applicable (e.g. package or library) for a data scientist to use in his daily work. Eventually, the state of the art concerning memory complexity estimation seems to be particularly sparse.

The proposed package `GuessCompx` (Agenis and Bokde, 2019) enables the R user to empirically estimate the computation time and memory usage of any algorithm before fully running it. As per author's knowledge, it is the first package in CRAN which discusses empirical estimation of algorithm complexity. The user's algorithm is run on an set of increasing-sizes small portions of his dataset. Various models are then fitted to capture the computational complexity trend likely to best fit the algorithm (independent *O(1)* , linear *O(N)* , quadratic *O(N^2)*, etc.), one for the time, another for the memory. The model eventually predicts the time and memory usage for the full size of the data. We restrict the complexity estimation to be only defined with regard to `N`, as it could indeed defined with regard to other parameters than the data size (think of data dimension, or number of features, of time horizon, etc.), and even interactions between those parameters, something that could be investigated in future versions of the package.

Details on the subject of algorithmic complexity can be found at Wikipedia contributors (2019).

The complexity functions already implemented are the following: *O(1), O(N), O(N^2), O(N^3), O(N^0.5), O(log(N)), O(N*log(N))*

`GuessCompx` package description

An asymptotic complexity behavior can be defined for most common algorithms: some are independent of the length of the data (think of the `length` function), some linear, some quadratic (typically a distance computation), etc. We track the computation time and memory of runs of the algorithm on increasing subsets of the data, using sampling or stratified sampling if needed. We fit the various complexity functions with a simple `glm()` procedure with a formula of the kind `glm(time ~ log(nb_rows))`, then find which is the best fit to the data. This comparison between the models is achieved through a LOO (**leave-one-out**) routine using Mean Squared Error as the indicator.

The `GuessCompx` package has a single entry point: the `CompEst()` function that accept diverse input formats (data.frame, matrix, time series) and is fully configurable to fit most use cases: which size of data to start at, how much time the user has to do the audit (usually 1 minute gives a good result), how many replicates you needed for each tested size (in case of high variability), is a stratified sampling required (in case each run must include all possible categories of one variable), by how much we increase the size at each run, etc.

The **plot output** helps to compare the fit of each complexity function to the data.

Note on the memory complexity: memory analysis relies on the `memory.size()` function to estimate the trend and this function only works on Windows machines. If another OS is detected, the algorithm will skip the memory part.

Imports

`GuessCompx` requires the following CRAN packages to be installed: **`dplyr`, `reshape2`, `lubridate`, `ggplot2`, `boot`**

Installation

The package can be downloaded from the CRAN repositories:

```
# install.packages("GuessCompx")
library(GuessCompx)
# ?CompEst
```

Also, it can be download it from Github:

```
# install.packages("devtools")
library(devtools)
install_github("agenis/GuessCompx")
```

Introduction to functions in `GuessCompx` package

Following sub-sections provide the insight on the main and internal functions developed in the `GuessCompx` package.

The `CompEst()` function:

`CompEst()` is the main function used for estimation of complexity of an algorithm. The function first creates a vector of data sizes, with possible replicates, and loops through it. The `max.time` argument is then used as a stopping condition to the loop (it stops as soon as the previous iteration has exceeded the time limit). Indeed, the `max.time` argument does not limit the total time spent by the function, but the maximum time of one single iteration including possible replicates: the total number of iterations will be high if the power factor is low, if the starting size is also low, and if the running time increases slowly. The user might have to think about how to set his parameters to prevent the computation time to be too long.

The core of the `CompEst` function works by simultaneously evaluating the time and the memory used by one iteration of the target function over a sample of the data. Time evaluation is achieved via a call to `system.time()`, memory evaluation is done through `memory.size()` function, called before and after a double garbage collection `gc()`. Bypassing the limitations of the `memory.size()` function to only Windows users has been unsuccessfully tried (`pryr::mem_change, Rprof`). Note that the sampling phase is done outside the time/memory evaluation, so it has no footprint on the results.

A dataframe containing the data sizes, the memory and time results, is created and serves as input to fit 2*7 complexity models and add their predictions to the data. The `cv.glm()` function eventually computes the Leave-One-Out RMSE error of each model, in an efficient way. Eventually, once the best model is selected (the one with the smallest LOO-error), a significance test is done to check if the model is better than an intercept-only model.

It must be noted that, when a CONSTANT relationship is predicted, it might simply mean that the `max.time` value is too low to show any tendency. Several such cases rise an alert to the user to suggest modifications in the value of the function's arguments.

The function returns a list with the best complexity model and the computation time on the whole dataset, for both time and memory complexity (Windows) and time complexity only (all other OS). It also returns two plots, with the best model curve highlighted in bold. The function has following arguments:

```
CompEst(d, f, random.sampling = FALSE, max.time = 30,
        start.size = NULL, replicates = 4, strata = NULL,
        power.factor = 2, alpha.value = 0.005, plot.result = TRUE)
```

`d`: A dataframe on which the algorithm is to be tested. This can also be a vector or a matrix.

`f`: A user-defined function that runs the algorithm. The algorithm takes `d` as the first argument. There is no need for the function to return any value.

`random.sampling`: This argument can have only two possible values, either TRUE or FALSE (boolean). The default value is set to FALSE. If the value is TRUE, a random sample is taken at each step. If False, the first N observations are taken at each step. Choosing a random sampling is relevant whith the use of replicates to help the discrimination power for complexity functions.

`max.time`: The maximum time allowed for each step of the analysis in seconds, that is the time for all replicates of a single sample size. Once the specified time limit is reached, the execution of the function is stopped. If no value is specified, the default value of 30 seconds is used. There is no such limitation regarding memory.

`start.size`: The size of the first sample to run the algorithm. The size is given in form of number of rows. The default value of the argument is `floor(log2(nrow(d)))`. If strata is not not NULL, it is recommended to pass a value which is multiple of number of categories.

`replicates`: The number of replicated runs of the algorithm for a specific sample size. This argument allows better discrimination of the complexity function. The default value of the argument is set to 2.

`strata`: This argument is a string containing the name of categorical column of d that must be used for stratified sampling. A fixed proportion of the categories will be sampled, always keeping at least one observation per category.

`power.factor`: The common ratio of the geometric progression of the sample sizes. The default value is 2. It means that sample sizes double every step. The argument can also be passed as a decimal value.

`alpha.value`: The alpha risk of the test whether the model is significantly different from a constant relation. The default valuesis set to 0.005.

`plot.result`: A boolean value to indicate if the summary plot of all the complexity functions is to be displayed. The best model is shown by the curve in bold. If FALSE, then the plot is not displayed. The default value is set to TRUE.

The `CompEstBenchmark()` function:

`CompEstBenchmark()` function presents a benchmark procedure to fit complexity functions to a dataframe of time or memory values. The function takes input a dataframe produced by the `CompEst()` function. User also needs to specify whether function deals with time or memory data. The function returns a list of all the fitted complexity model.

The function has following arguments:

```
CompEstBenchmark(to.model, use = "time")
```

`to.model`: A dataframe produced by the `CompEst()` function. The dataframe is comprised of size, time, memory and *NlogN_X* columns.

`use`: A string indicating if the function deals with time or memory data. The default value of the argument is “time”.

The `CompEstPlot()` function:

The `CompEstPlot()` function plots the results of algorithms complexities. It returns a “ggplot” object.

The arguments of the function are:

```
CompEstPlot(to.plot, element_title = list("", ""), use = "time")
```

`to.plot`: A dataframe produced by `CompEst()` function.

`element_title`: A string that will be added to the subtitle of the plot.

`use`: A string to indicate if the function deals with “time” or “memory” data. The default value is “time”.

The `CompEstPred()` function:

The `CompEstPred()` function predicts the computation time of a whole dataset. The function returns the predicted time for the whole dataset.

The arguments are:

```
CompEstPred(model.list, benchmark, N, use = "time")
```

`model.list`: A list containing the fitted complexity functions, produced by `CompEst()` function.

`benchmark`: A vector of LOO errors of complexity functions, produced by `CompEst()`.

`N`: Number of rows of the whole dataset, produced by `CompEst()`.

`use`: A string indicating if the function deals “time” or “memory” data.

`GroupedSampleFracAtLeastOneSample()` and `rhead()`

The `GroupedSampleFracAtLeastOneSample()` function samples a random proportion of data, keeping at least one observation. This function is designed to allow its use with group splitting or do.by methods. The function takes as input a dataframe from which a small sample is to be returned. It is also possible to specify the desired sampling fraction (second argument). The value of sampling

fraction is between 0 and 1. The third argument, `is.random`, is a boolean value. If TRUE, a random sample is drawn, else it takes the `head()` of the data.

The `rhead()` generates small random samples from a vector or a dataframe. The function takes input a vector or a dataframe from which the small random samples are generated. The second argument is a positive integer, representing the number of lines or elements to print. The default value is 7. The third argument, `is.random` is same as the one of the above functions. All functions other than `CompEst` and `rhead()` are internal and not directly accessible to users, but corresponding codes are available at Github page (Agenis, 2019b).

Demonstration of `Guesscompx` package

This section demonstrates the usefulness of the `Guesscompx` package. It provide several examples showing some use cases (distance function, time series, custom function) and how to best tune the parameters. It is important to keep in mind, since this package is measuring actual CPU times, absolute reproducibility is out of reach. Running the examples on the computer will return slightly different results each time.

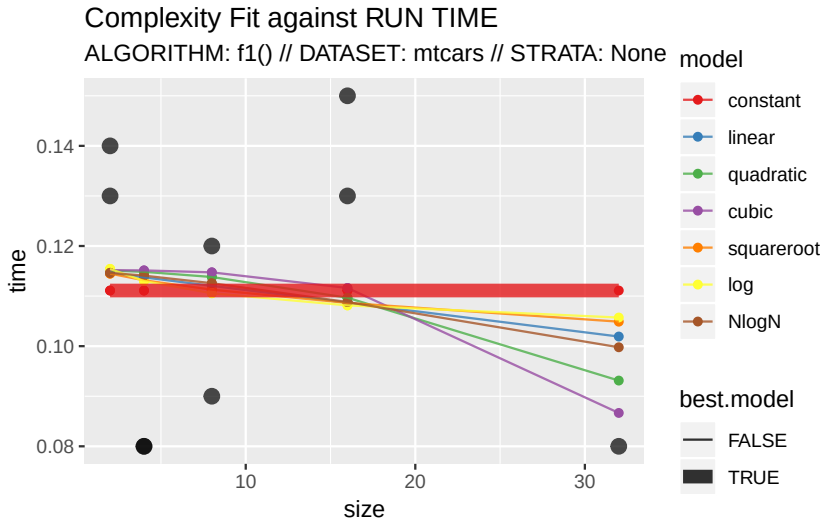
Example 1

This example sets up a dummy function that mimics an algorithm with a *O(1)* time complexity and a *O(N.log(N))* memory complexity, both with some random noise. See the warning issued for the time complexity : when a constant model is found, it always suggests that the cause might be insufficient running time or insufficient replicates. Also note that a *O(N.log(N))* trend can sometimes be mistaken for a linear trend.

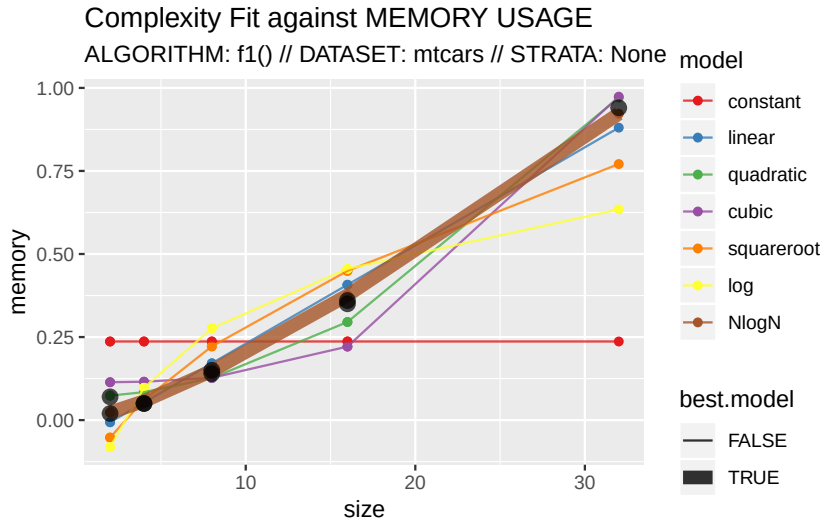
```
# Dummy function that mimics a constant time complexity and
# N.log(N) memory complexity:
f1 = function(df){
  Sys.sleep(rnorm(1, 0.1, 0.02))
  v = rnorm(n = nrow(df)*log(nrow(df))*(runif(1, 1e3, 1.1e3)))
}
out = CompEst(d = mtcars, f = f1, replicates=2, start.size=2, max.time = 1)

# Raises an alert for TIME complexity.
# Sometimes confuses MEMORY complexity with linear:
print(out)
```

```
#> $sample.sizes
#>  [1]  2  2  4  4  8  8 16 16 32 32
#>
#> $`TIME COMPLEXITY RESULTS`
#> $`TIME COMPLEXITY RESULTS`$best.model
#> [1] "CONSTANT"
#>
#> $`TIME COMPLEXITY RESULTS`$computation.time.on.full.dataset
#> [1] "0.11S"
#>
#> $`TIME COMPLEXITY RESULTS`$p.value.model.significance
#> [1] NA
#>
#>
#> $`MEMORY COMPLEXITY RESULTS`
#> $`MEMORY COMPLEXITY RESULTS`$best.model
#> [1] "NLOGN"
#>
#> $`MEMORY COMPLEXITY RESULTS`$memory.usage.on.full.dataset
#> [1] "1 Mb"
#>
#> $`MEMORY COMPLEXITY RESULTS`$system.memory.limit
#> [1] "8064 Mb"
#>
#> $`MEMORY COMPLEXITY RESULTS`$p.value.model.significance
#> [1] 3.166196e-09
```



(a)



(b)

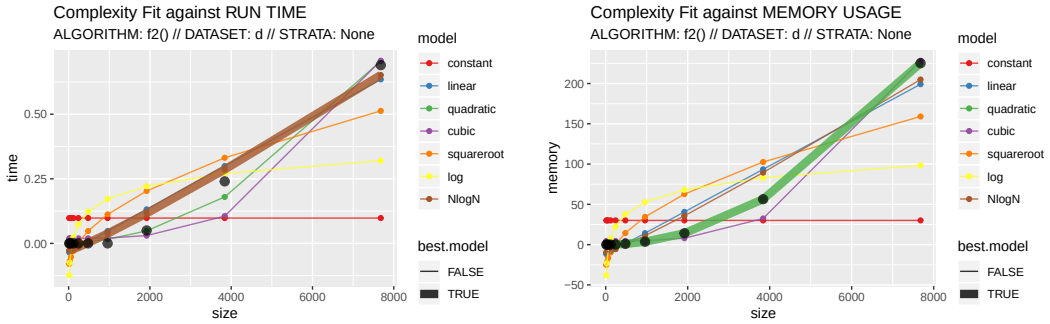
Figure 1: Complexity fit against (a) run time and (b) memory usage for function 'f()' on 'mtcars' dataset, suggests *O(1)* and *O(Nlog(N))* as best model, respectively (Example 1)

Example 2

This example tests the behaviour against a real-life distance algorithm whose complexity should be a clear *O(N^2)*, for both memory and time.

```
# 'dist' function analysis:
f2 = dist
d  = ggplot2::diamonds[, 5:8]
CompEst(d = d, f = f2, replicates = 1, max.time = 1)
```

```
#> $sample.sizes
#>  [1]    15    30    60   120   240   480   960  1920  3840  7680 15360
#> [12] 30720 53940
#>
#> $`TIME COMPLEXITY RESULTS`
#> $`TIME COMPLEXITY RESULTS`$best.model
#> [1] "NLOGN"
#>
#> $`TIME COMPLEXITY RESULTS`$computation.time.on.full.dataset
#> [1] "5.78S"
```



(a) **(b)**

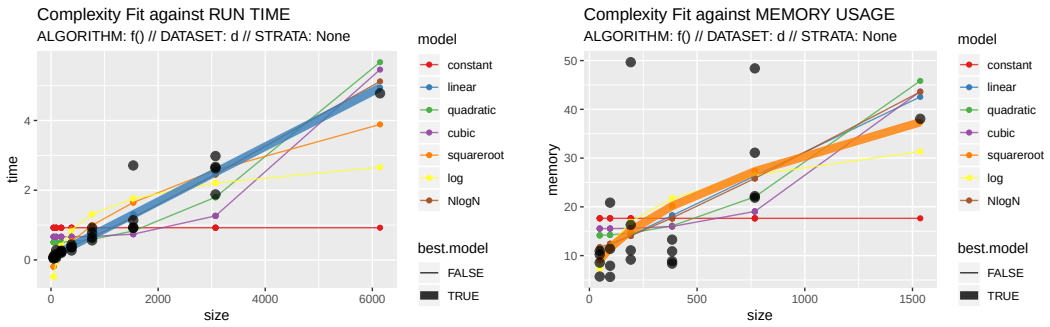
Figure 2: Complexity fit against (a) run time and (b) memory usage for distance function on 'diamonds' dataset, suggests *O(Nlog(N))* and *O(N^2)* as best model, respectively (Example 2)

```
#>
#> $`TIME COMPLEXITY RESULTS`$p.value.model.significance
#> [1] 1.57405e-07
#>
#>
#> $`MEMORY COMPLEXITY RESULTS`
#> $`MEMORY COMPLEXITY RESULTS`$best.model
#> [1] "QUADRATIC"
#>
#> $`MEMORY COMPLEXITY RESULTS`$memory.usage.on.full.dataset
#> [1] "11110 Mb"
#>
#> $`MEMORY COMPLEXITY RESULTS`$system.memory.limit
#> [1] "8064 Mb"
#>
#> $`MEMORY COMPLEXITY RESULTS`$p.value.model.significance
#> [1] 1.28452e-28
```

Example 3

This example tests the time and memory complexity of a time series prediction method. For this purpose, an ARIMA model is used. For time series functions, the `f` argument may include `ts()` and to avoid loosing this `ts` attribute at sampling, it is recommended to set `start.size` argument to 3 periods at least. The ARIMA function should return a linear trend for time complexity.

```
# time series prediction function analysis:
f = function(d) arima(ts(d, freq = 12), order=c(1,0,1), seasonal = c(0,1,1))
d = ggplot2::txhousing$sales
CompEst(d, f, start.size = 4*12, random.sampling = FALSE)
```



(a) **(b)**

Figure 3: Complexity fit against (a) run time and (b) memory usage for distance function on 'sales' dataset, suggests *O(N)* and *O(N^(1/2))* as best model, respectively (Example 3)

```
#> $sample.sizes
#>  [1]   48   48   48   48   96   96   96   96  192  192  192  192  384  384
#> [15]  384  384  768  768  768  768 1536 1536 1536 1536 3072 3072 3072 3072
#> [29] 6144 6144 6144 6144 8602 8602 8602 8602
#>
#> $`TIME COMPLEXITY RESULTS`
#> $`TIME COMPLEXITY RESULTS`$best.model
#> [1] "LINEAR"
#>
#> $`TIME COMPLEXITY RESULTS`$computation.time.on.full.dataset
#> [1] "6.88S"
#>
#> $`TIME COMPLEXITY RESULTS`$p.value.model.significance
#> [1] 3.492077e-16
#>
#>
#> $`MEMORY COMPLEXITY RESULTS`
#> $`MEMORY COMPLEXITY RESULTS`$best.model
#> [1] "SQUAREROOT"
#>
#> $`MEMORY COMPLEXITY RESULTS`$memory.usage.on.full.dataset
#> [1] "84 Mb"
#>
#> $`MEMORY COMPLEXITY RESULTS`$system.memory.limit
#> [1] "8064 Mb"
#>
#> $`MEMORY COMPLEXITY RESULTS`$p.value.model.significance
#> [1] 0.004649934
```

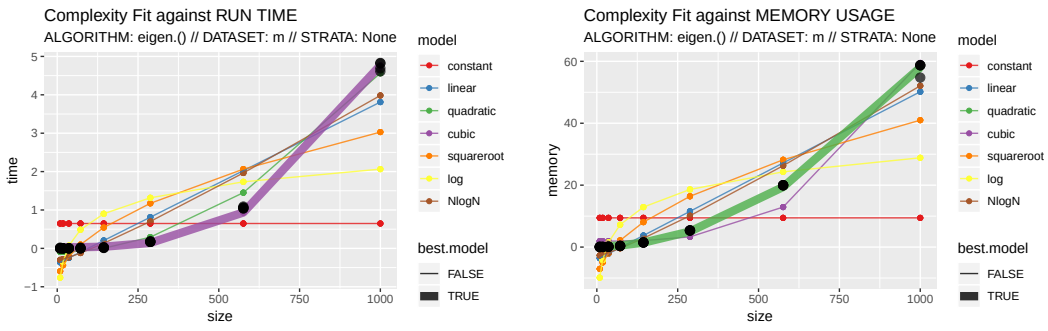
Example 4

This example tests the behavior of eigen decomposition process of a matrix. Since the algorithm only works on square matrices, here the target function subsets columns to force the matrix into a square one, otherwise the sampling would result in a rectangular matrix. This analysis yields an *O(N^3)* fit for time complexity, which is scientifically correct, and a quadratic for memory one.

```
# Eigendecomposition of a matrix
m = matrix(rnorm(1e6), ncol=1000, nrow=1000)

# force the matrix into a square one:
eigen. = function(m) eigen(as.matrix(m[, 1:nrow(m)]))

# This yields an O(n^3) fit, which is scientifically correct,
# and a quadratic for memory.
out = CompEst(m, eigen., replicates = 5, max.time = 60)
```



(a) **(b)**

Figure 4: Complexity fit against (a) run time and (b) memory usage for eigen decomposition function on a matrix dataset, suggests *O(N^3)* and *O(N^2)* as best model, respectively (Example 4)

Example 5

This example is an special case of stratified input, which is useful to force the sampling to have at least one observation of each class of a specific column. Consider a function to predict a diabetes outcome in the dataset 'PimaIndiansDiabetes' with the Support Vector Machine (SVM) method. Depending on algorithm, the true complexity (time) of SVM is between *O(N^2)* and *O(N^3)*.

```
library(mlbench)
library(e1071)
data("PimaIndiansDiabetes")
f6 = function(df){
  fit = svm(diabetes ~ ., data=df)
  return(table(df$diabetes, fitted(fit)))
}
```

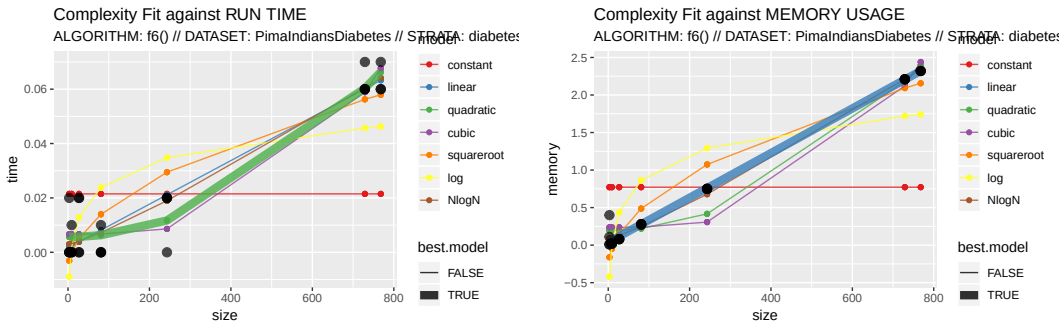
With this configuration, sampling will often result in only negative or positive observations to train the model, which results in an error message as shown below.

```
CompEst(PimaIndiansDiabetes, f6, start.size = 3, power.factor = 3)

# ERROR: model is empty!
```

So, after fixing the strata argument to be the Y vector of classes, to ensure that both classes will be represented in the model as follows. Currently, the strata argument enables to specify only one column, but future versions of this package will accept list of columns.

```
CompEst(PimaIndiansDiabetes, f6, start.size = 3, power.factor = 3, strata = "diabetes")
```



(a) **(b)**

Figure 5: Complexity fit against (a) run time and (b) memory usage for SVM function on 'PimaIndiansDiabetes' dataset, suggests *O(N^2)* and *O(N)* as best model, respectively (Example 5)

```
#> $sample.sizes
#>  [1]   3   3   3   3   9   9   9   9  27  27  27  27  81  81  81  81 243
#> [18] 243 243 243 729 729 729 729 768 768 768 768
#>
#> $`TIME COMPLEXITY RESULTS`
#> $`TIME COMPLEXITY RESULTS`$best.model
#> [1] "QUADRATIC"
#>
#> $`TIME COMPLEXITY RESULTS`$computation.time.on.full.dataset
#> [1] "0.07S"
#>
#> $`TIME COMPLEXITY RESULTS`$p.value.model.significance
#> [1] 8.624223e-15
#>
#>
#> $`MEMORY COMPLEXITY RESULTS`
#> $`MEMORY COMPLEXITY RESULTS`$best.model
#> [1] "LINEAR"
#>
#> $`MEMORY COMPLEXITY RESULTS`$memory.usage.on.full.dataset
```

```
#> [1] "2 Mb"
#>
#> $`MEMORY COMPLEXITY RESULTS`$system.memory.limit
#> [1] "8064 Mb"
#>
#> $`MEMORY COMPLEXITY RESULTS`$p.value.model.significance
#> [1] 7.60052e-29
```

Example 6

This example tests the complexity of a dummy function that mimics a linear time complexity with high CPU variability and heteroscedasticity. This function results in a variety of non-significant "best models", often "LINEAR" one. Thereafter, we increase the number of replicates, start at a smaller size, and get an almost always "Linear" response. This example takes much more time because we replicate 10 times the result.

```
# A dummy function:
f3 = function(df){  Sys.sleep(max(0, rnorm(1, 0.1+nrow(df)/500, nrow(df)/2000)))  }
set.seed(1);
replicate(10, CompEst(d = mtcars, f = f3, plot.result = F)$`TIME COMPLEXITY RESULTS`$best.model)

#>  [1] "LOG"        "SQUAREROOT" "LINEAR"     "LOG"        "SQUAREROOT"
#>  [6] "QUADRATIC"  "SQUAREROOT" "QUADRATIC"  "SQUAREROOT" "SQUAREROOT"

set.seed(2);
replicate(10, CompEst(d = mtcars, f = f3, plot.result = F, start.size = 1,
replicates = 10, max.time=5)$`TIME COMPLEXITY RESULTS`$best.model)

#>  [1] "SQUAREROOT" "LINEAR"     "NLOGN"      "SQUAREROOT" "LINEAR"
#>  [6] "LINEAR"     "LINEAR"     "LINEAR"     "NLOGN"      "NLOGN"
```

Performance

As in (Sharma et al., 2018), we tried to assess the predictive power of the described method. It is important to understand that prediction performance of our tool has no absolute meaning, since it can always be improved by allocating more time to run the test.

Table 1: Some famous algorithms with their predicted complexities

	True time complexity	Time Accuracy	True space complexity	Predicted space complexity
Bubble sort	O(N^2)	97%	O(N)	97% O(Nlog(N))[1]
Find max	O(N)	95%	O(1)	65% O(1)
Permutations	O(N^3)	99%	1	95% O(N)
Tree split[2]	O(LOG(N))	38%	1	98% O(NlogN)
Shell search	O(N^(4/3))[3]	NA (73% O(NlogN), 27% O(N))	1	100% O(N)

These considerations taken aside, a test bench was set up in order to test one paragon algorithm for each complexity family as shown in Table 1. Whenever possible, we worked with the default setting of the `CompEst()` function, sometimes reducing the time limit down to 1 second. One hundred replicates of the function were set. Accuracy is defined as:

$$\text{Accuracy} = \frac{\text{Number of correct predictions for complexity function}}{\text{total number of cases}} \times 100$$

Eventually, some of the precision results may seem disappointing. There is indeed a strong sensibility to the choice of the settings and fine-tuning the arguments can make a sensible difference in the result. It is illustrated by a detailed simulation where the maximum vector length varies between

[1] it is known that linear and *O(NlogN)* are easy to confuse
[2] rpart regression from rpart package, which is sensitive to the data itself
[3] not part of the 7 base complexity functions

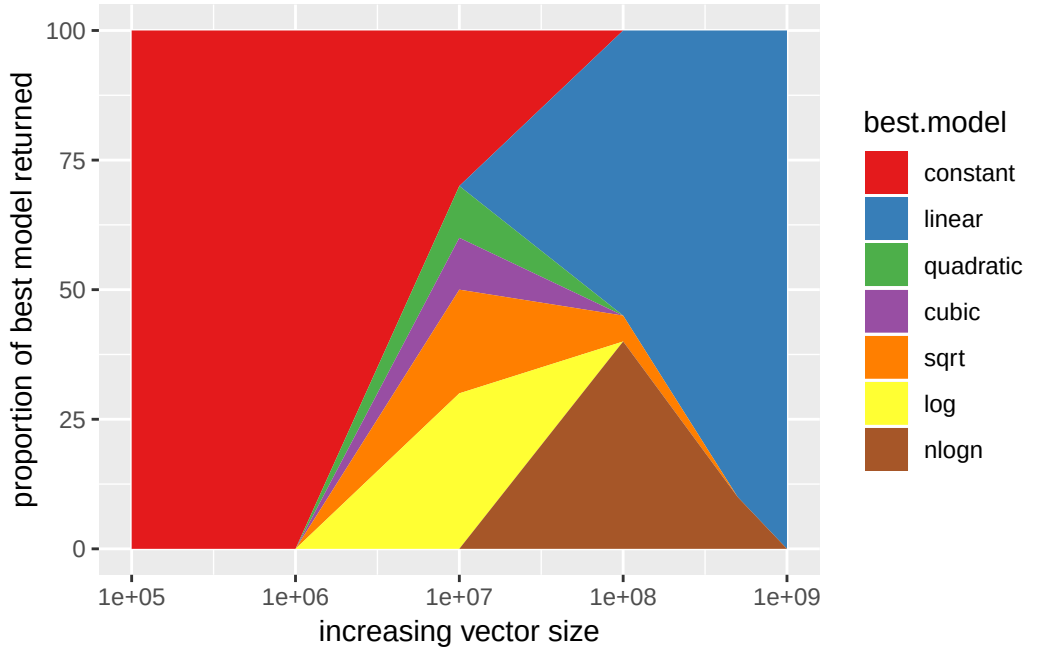


Figure 6: The asymptotic linear trend for 'max()' function as target algorithm

1E4 and 1E9, this last value being considered as the "asymptotic" value. The Figure 6 shows asymptotic linear trend for 'max()' function as target algorithm. It represents the result in proportion of the best model outputted: when the data is too small no model can be determined because the 'max()' function is too fast; for vectors around size 1E07, CPU time starts to rise with some variability, but the fit will not return a stable solution; eventually, with sizes going over 1E08, the asymptotic linear trend appears strongly.

Number of Replicates:

As a second performance example, the number of replicates is tested in the benchmark. The `max.time` limit argument is not used here because it is counted for all replicates of the same size. In order to hold everything constant but the number of replicates, we set an infinite time limit for a run, only limiting it with a small input dataset, so it will will not take more than a fraction of seconds to complete a run. This testbench, using a distance computation on a data.frame as the target algorithm, shows that when the we are in the "grey zone" (uncertain outcome of the analysis), the number of replicates can make the difference to find the correct model as shown in Figure 7.

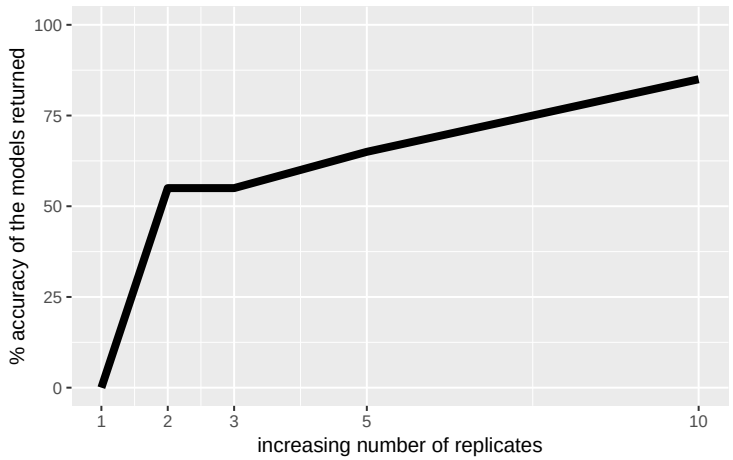


Figure 7: Effect of number of replicates on finding the correct model for distance algorithm on 'diamonds' dataset

The codes to test all analysis and comparison done in this section are available in GitHub page (Agenis, 2019a).

Summary

This article is a thorough description of the `GuessCompx` R package. This package is introduced to makes an empirical guess on the time and memory complexities of an algorithm or a function. It tests multiple, increasing-sizes samples of the user's data and try to fit one of seven complexity functions: *O(N), O(N^2), O(log(N))*, etc. Based on a best fit procedure using LOO-MSE (leave one out-mean square error), it also predicts the full computation time and memory usage on the whole dataset. The hereby suggested method and package are believed to be new to the R users community; however there is a lot of room for improvement, both in terms of automation and variety of complexity functions.

Bibliography

M. Agenis. Guesscompxperformancetests. https://github.com/agenis/GuessCompxPerformanceTests, 2019a. [p12]

M. Agenis. Guesscompx. https://github.com/agenis/GuessCompx, 2019b. [p5]

M. Agenis and N. Bokde. *GuessCompx: Empirically Estimates Algorithm Complexity*, 2019. URL https://CRAN.R-project.org/package=GuessCompx. R package version 1.0.3. [p2]

I. Chivers and J. Sleightholme. An introduction to algorithms and the big o notation. In *Introduction to Programming with Fortran*, pages 359–364. Springer, 2015. [p1]

S. F. Goldsmith, A. S. Aiken, and D. S. Wilkerson. Measuring empirical computational complexity. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 395–404. ACM, 2007. [p1]

J. He and X. Yao. Drift analysis and average time complexity of evolutionary algorithms. *Artificial intelligence*, 127(1):57–85, 2001. [p1]

M. T. Jensen. Reducing the run-time complexity of multiobjective eas: The nsga-ii and other algorithms. *IEEE Transactions on Evolutionary Computation*, 7(5):503–515, 2003. [p1]

W. Min. Analysis on bubble sort algorithm optimization. In *2010 International Forum on Information Technology and Applications*, volume 1, pages 208–211. IEEE, 2010. [p1]

J. Qiu, Q. Wu, G. Ding, Y. Xu, and S. Feng. A survey of machine learning for big data processing. *EURASIP Journal on Advances in Signal Processing*, 2016(1):67, 2016. [p1]

D. K. Sharma, S. Vohra, T. Gupta, and V. Goyal. Predicting the algorithmic time complexity of single parametric algorithms using multiclass classification with gradient boosted trees. In *2018 Eleventh International Conference on Contemporary Computing (IC3)*, pages 1–6. IEEE, 2018. [p1, 2, 10]

L. G. Valiant. The complexity of computing the permanent. *Theoretical computer science*, 8(2):189–201, 1979. [p1]

Wikipedia contributors. Time complexity — Wikipedia, the free encyclopedia, 2019. URL https://en.wikipedia.org/wiki/Time_complexity. [Online; accessed 01-May-2019]. [p2]

G. J. Woeginger. Space and time complexity of exact algorithms: Some open problems. In *International Workshop on Parameterized and Exact Computation*, pages 281–290. Springer, 2004. [p1]

Y. Yang, P. Yu, and Y. Gan. Experimental study on the five sort algorithms. In *2011 Second International Conference on Mechanic Automation and Control Engineering*, pages 1314–1317. IEEE, 2011. [p1]

Marc Agenis-Nevers
Independent Researcher
Epicentre Factory, Clermont-Ferrand
France
marc.agenis@gmail.com

Neeraj Dhanraj Bokde
Visvesvaraya National Institute of Technology, Nagpur
North Ambazari Road, Nagpur
India
neerajdhanraj@gmail.com

Mayur Shende
Government College of Engineering, Nagpur
India
mayur.k.shende@gmail.com