PARALLELIZATION OF SUBSET SELECTION IN LINEAR REGRESSION

Abstract

Parallelization of subset selection for multivariate linear regression using R packages doParallel and foreach

Hosmani, Chaitra
Student Id: 500800789
Chaitra.Hosmani@Ryerson.ca

Contents

1. Introduction

In machine learning linear regression is popular modeling method. We predict the response variable given a set of input variables(features). As number of input variables increases, the model can become complex, increases computation and memory usage and difficult to comprehend. It will also be more susceptible to noise. It is necessary to exclude some features as they might be adding less value in predicting response variable. Hence, we use dimensionality reduction to explain the model which reduced number of features. This helps to make the model simple and comprehensible.

This paper uses subset selection as the way to reduce the dimensions. Given N number of dimensions(features), we will have 2^N combinations of models to choose from. i.e. we evaluate the performance of each model and choose the best fitting model. This is method is not scalable and will perform poorly as number of features increases. It is essential to paralyse the algorithm to improve the scalability. Hence, this paper gives the method to paralyze subset selection procedure for multiple linear regression.

2. Goal

Implement subset selection in multiple regression using R with libraries *foreach* and *doParallel*. Compare the efficiency of algorithm in linear versus parallel algorithms.

Design sequential algorithm using R which find the best model with subset of features. Paralyze the algorithm with doParallel and foreach packages. Goal is increase the efficiency by utilising the given resource to the fullest.

3. Related Literature

[2] In this paper, the approach of paralysation is using map reduce. Since the data and number of features is large, each worker node can execute the program parallelly to compute the model. This is scalable as task gets distributed among various worker nodes and make use of individual machine's computation power. Hence the theoretical increase of speed up can total_time_in_linear_execution/Number of parallel process. This method requires setting up the Hadoop system with large number of worker nodes.

[3] This paper has the bi-objective approach to find the best fitting model. i.e. it finds the best model by continuously decreasing the unwanted features and increasing the performance by adding the features. It produces the paralyse version of Pareto Optimization for Subset Selection

[4]In this paper, feature subset selection program is using parallel scatter search in order to reduce the execution time. The paper focuses on simplicity of the solution and keeping the time to run the parallel program as same as the sequential one. Since it's involves greedy method, it provides the solution which is closer to optimal solution

4. Method

Data Preprocessing: The data file is named as "sample_data" and all input variables are named as x1,x2,x3 so on so forth. The output variable is named as y.

Algorithm

If N is the number of given features and we want to reduce our model with at least k features where k <N. We will start with k=1 and find the best model for single variable. Keep incrementing k and find the best model at each k. This will be an exhaustive approach which outputs the performance of all combination for given k.

lm function: used for fitting linear regression model. This intake the response variable and input variables as first argument, data as the second argument. Returns the coefficients and Intercept. The summary for lm will returns R square, adjusted R square and many others values

combn function: will provide all the combinations of variable for given k

Choosing the best model: Best model as each level of k is selected based on adjusted R square. This gives better result than using R square, as R square will always increase as number of variables are increased. Human judgement is also required to finalise the best model among different k as it's a trade off between simple model and optimized solution.

Since this is exhaustive approach and and as N increases the number of combinations to test increases exponentially. We will restrict test the results with k at 15

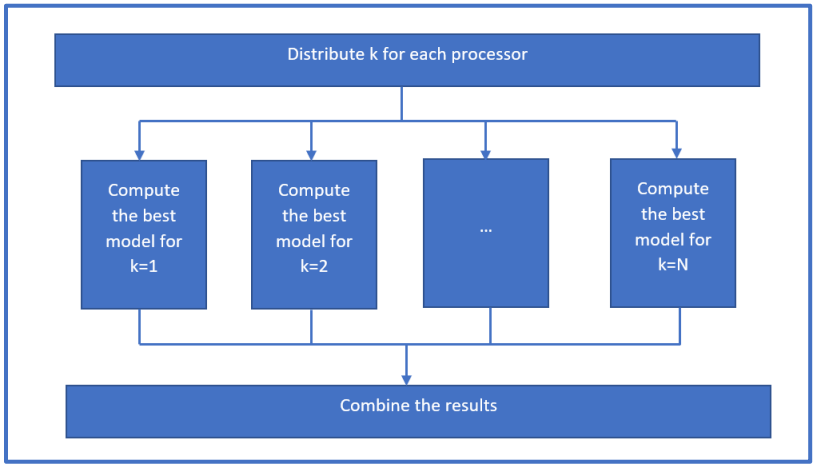
pseudocode for subset selection algorithm

```
Initialization
    Input data and initialize k,n, iterations
Iteration
    # for each k
    for ( k =1 to N) do
        #Get all the combinations of k
        combinations=combn(x, k)
        # for each combination
        for (i =1 to combinations)
            #find the result of model with adjusted R square
            # use the lm function in R
             current_adjusted_r_square = summary(lm(i[current_combination], data=sample_data))$adj.r.squared
            # compare the results with previous higest adjusted_r_square
            if (current_adjusted_r_square > adjusted_r_square)
                choose the best model as current model
                adjusted_r_square = current_adjusted_r_square
        output the best model for given k
```

Parallelization

Since, each combination of k can be implemented independently, we should make use of available processors of the current hardware. We can easily split the job on different nodes, each node executing the different values of k and combining the result at the end. Below picture depicts the flow of parallel programming

[1] The doParallel package is a "parallel backend" for the foreach package. It provides a mechanism needed to execute foreach loops in parallel. The foreach package must be used in conjunction with a package such as doParallel in order to execute code in parallel. The user must register a parallel backend to use, otherwise foreach will execute tasks sequentially, even when the %dopar% operator is used.



5. Results

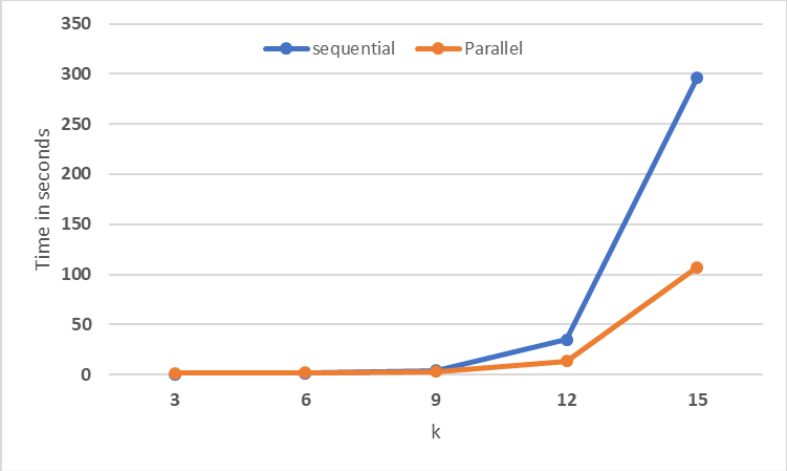
The program is executed against various combinations of k and N. Following are some of the output.

A. Constant Number of instances and variable k:

Keeping the N=21263 as constant the program is executed for different values of k

#Instance	21263	
k	**Sequential time**	**Parallel time**
3	0.3293972 secs	1.400219 secs
6	1.306427 secs	1.835298 secs
9	4.003044 secs	3.232083 secs
12	34.66834 secs	13.30102 secs
15	4.938223 mins	1.788039 mins

Graph below shows the comparison between sequential and parallel algorithm

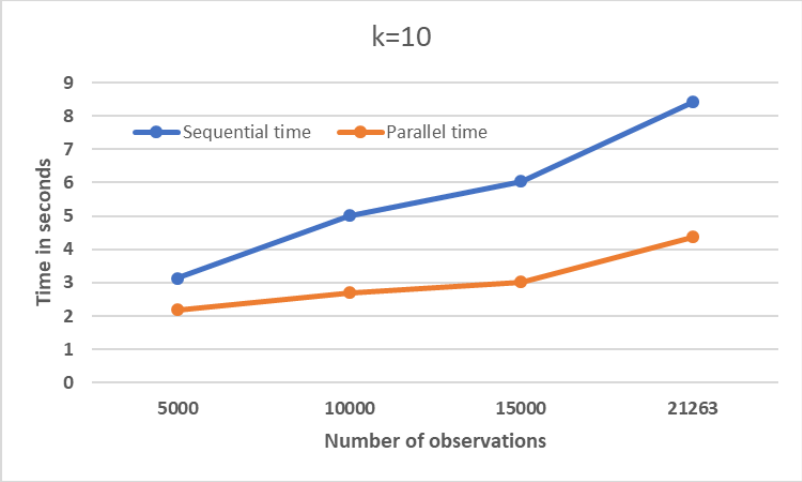


B. Constant k and variable #of instances:

I. Keeping the k as 10, the program is run for 5000, 10000, 150000 and 21263 number of instances

k	10	
#of instance	**Sequential time**	**Parallel time**
5000	3.136759 secs	2.188092 secs
10000	5.011494 secs	2.699435 secs
15000	6.044075 secs	3.015062 secs
21263	8.409596 secs	4.366186 secs

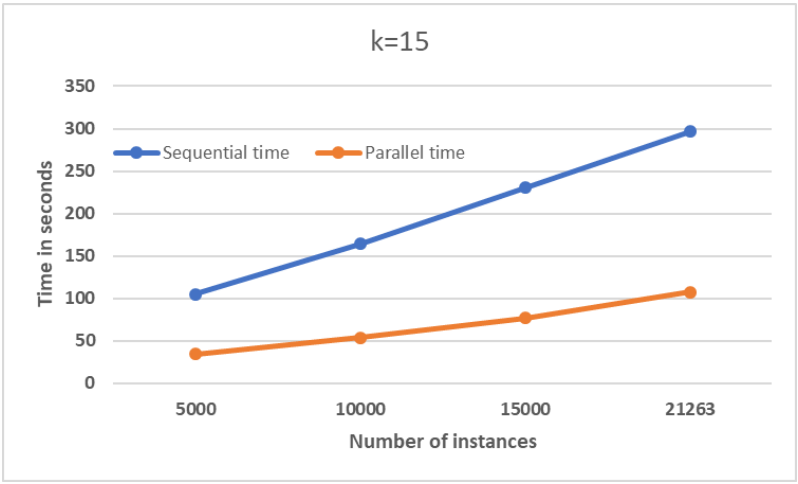
Graph below shows the comparison between sequential and parallel algorithm



II. Keeping the k as 15, the program is run for 5000, 10000, 150000 and 21263 number of instances

k	15	
#of instance	**Sequential time**	**Parallel time**
5000	1.747029 mins	34.30154 secs
10000	2.737507 mins	53.70812 secs
15000	3.835412 mins	1.279494 mins
21263	4.938223 mins	1.788039 mins

Graph below shows the comparison between sequential and parallel algorithm



6. Summary

Constant #of instances and variable k

Based on the result we can infer that if smaller number k, sequential program is more efficient than parallel algorithm as it includes the overhead cost associated with inter communication. Where as, when k tends to be larger value parallel algorithm proved to be more efficient than sequential.

Constant k and variable #of instances

Even though the time consumed as number of instances is increased, it does not increase exponential like above result. There is a linear increase in time when k kept as constant.

Hence, if k is small, parallelization may not be much helpful as there might be overhead cost associated with paralysation. If k is large, it is recommended to paralyze even when N is small.

If the number of variables is small, the time gain is not substantial as it includes the overhead cost of setting up the penalization. As the number of variables increase, parallel algorithm performs much better than sequential one.

7. References

1. [1] Getting started with doParallel and for each. Steve Weston and Rich Calaway
2. [2] http://additivegroves.net/papers/chapter-featureeval.pdf
3. [3] https://pdfs.semanticscholar.org/a1d4/3002927f90003c4176f547521990f1427c7a.pdf
4. [4] http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.97.1811&rep=rep1&type=pdf

8. Appendix

CODE1: Linear programming

```
#Subset selection linear programming
start_time <-Sys.time()

N=15  # number of features to be choosen
x=names(sample_data)[1:N]
k=1
models=list()
Error=c()
best_model=c()

for(k in 1:N)
{
 # finding all possible combinations for given k
   combinations=matrix()
   combinations=combn(x, k)
   all_comb = dim(combinations)[2]
   i=1
   best_model=0
   min_adj_r_sq =-9999999999;
   # loop through all the combinations of k
   for(i in 1:all_comb)
   {
      independent_var= combinations[,i]
      j=1
      lm_arg=""
      # create lm argument by looping through all the varibale of a given combination
      for(j in 1:k)
      {
         if(lm_arg != "")
         {
            lm_arg = paste(lm_arg, paste0("sample_data$", independent_var[j]), sep="+")
         }
         else
         {
            # when combination has only one variable
            lm_arg = paste0("sample_data$",independent_var[j] )
         }

      }
      # add the dependent variable to lm_arg
      lm_arg = paste0("sample_data$y~", lm_arg)
```

```
   # call the linear model function and adjusted R Square
   cur_adjr_sq = summary(lm(as.formula(lm_arg), data=sample_data))$adj.r.squared
   if(cur_adjr_sq > min_adj_r_sq)
   {

      best_model = i
      min_adj_r_sq = cur_adjr_sq
   }
  }  # end of all combinations for given k (i)
  print(combinations[,best_model])
}  # end of all k
end_time <-Sys.time()
# Total execution time
end_time - start_time
```

CODE#2 Parallel Programming

```
library(foreach)
library(doParallel)
# initializing the start time
start_time <-Sys.time()
# initializing number of cores
no_cores <- detectCores() - 1
cl <- makeCluster(no_cores)
registerDoParallel(cl)

N=15  # number of features to be choosen
x=names(sample_data)[1:N]
k=1
models=list()
best_model=c()

foreach(k = 1:N) %dopar%
{
   combinations=matrix()    # resetting the combinations for each round of k
   combinations=combn(x, k)
   all_comb = dim(combinations)[2]
   i=1
   # loop through all the combinations of k
   for(i in 1:all_comb)
   {
      min_adj_r_sq =-9999999999;
      independent_var= combinations[,i]
      j=1
```

```
        lm_arg=""
         # create lm argument by looping through all the varibale of a given combination
        for(j in 1:k)
        {
          if(lm_arg != "")
          {
            lm_arg = paste(lm_arg, paste0("sample_data$", independent_var[j]), sep="+")
          }
          else
          {
            # when combination has only one variable
            lm_arg = paste0("sample_data$",independent_var[j] )
          }

        }
      # add the dependent variable to lm_arg
      lm_arg = paste0("sample_data$y~", lm_arg)

      # call the linear model function and adjusted R Square
      cur_adjr_sq = summary(lm(as.formula(lm_arg), data=sample_data))$adj.r.squared
      if(cur_adjr_sq > min_adj_r_sq)
      {
        best_model = i
        min_adj_r_sq = cur_adjr_sq
      }

    }  # end of all combinations for given k (i)
    print(combinations[,best_model])
  }  # end of all k

on.exit(stopCluster(cl))
end_time <-Sys.time()
# Total execution time
end_time - start_time
```