*Multiple Regression, Backward Elimination pre-processing with Java*

*Anton Kovalyov*

*Brownsville, USA*

*Dr. Hansheng Lei*

*Brownsville, USA*

**The goal of this research is coming up with a program that takes different data as input and outputs all possible maximum size linear multiple regression functions. These functions must consist only of relevant variables, and the degree of multicollinearity in each function cannot be too high. This “pre-processing procedure” can help detect relations among different variables, apart from greatly reducing the computation time and space for later backward elimination procedures.**

INTRODUCTION

It is a known fact that including independent variables that have no relation with the dependent variable in multiple regression functions will only increase the variance of predictions [1], thus the predictors in a multiple regression function must have some type of relation with the independent variable. This relation can be set with some threshold T1.

Additionally, the estimates of regression coefficients can be highly unstable due to the presence of multiple predictors that have the same relationship with the outcome variable (multicollinearity), meaning that these predictors are highly correlated to each other. Dropping such predictors can increase the bias of the predictions, yet at the same time it will reduce the prediction variance, resulting in more accurate predictions overall. This is a common problem in data mining, known as the *bias-variance trade-off* [1]. Yet, it is a known fact that a high degree of multicollinearity can result in data redundancy and consequent overfitting in regression analysis models. As an extreme example, when predicting how overweight a person is, would it make sense to include both weight in kilograms and pounds as the predictor variables? Of course not!

To date, there are several articles that talk about detection of multicollinearity and different approaches on how to deal with it in multiple regression functions. This project proposes a different method: an algorithm that completely avoids the formation of multiple regression functions with high degrees of multicollinearity by setting a correlation threshold T2 among the independent variables that will exclude the addition of predictors, in a given multiple regression function, that are highly correlated to each other. Please note that for simplicity sake we assume T2 = T1 throughout the text.

So, in simpler words, the output of the program must be a set of all possible maximum size multiple regression functions, where the dependent variable has some correlation with the predictor variables while at the same time none of the predictor variables has a high correlation to each other. For example, let’s consider the following correlation matrix A, where T stands for true (there is a correlation between variablesxi and xj ) and F stands for false (there is no correlation between variables xi and xj):

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | X0 | X1 | X2 | X3 | X4 |
| X0 | T | F | T | F | T |
| X1 | F | T | F | T | T |
| X2 | T | F | T | T | T |
| X3 | F | T | T | T | F |
| X4 | T | T | T | F | T |

Fig. 1. Sample correlation matrix

The output should be the following:

X0: X2

X0: X4

X1: X3, X4

X2: X0, X3, X4

X3: X1, X2

X4: X0, X1

X4: X1, X2

Where the variables on the left represent the dependent variable (y) and those on the right, after the colon, represent the independent variables (x).

As you can see there are 7 functions, and in each function, y is correlated to the independent variables and the independent variables are non-correlated to each other.

Additionally, each set contains the maximum amount of dependent variables non-correlated-to-each-other as possible, and thus the functions are ready for the backward elimination algorithm to be applied on each to find those that result in most accurate predictions.

# METHODOLOGY

Before going to the explanation on how the algorithm works, there are a few steps that must be done first:

1. Find a dataset with a huge number of attributes.

For this part we used the dataset from <https://www.huduser.gov/portal/datasets/hads/hads.html>. This one, is a dataset about housing affordability in the United States and it consists of 99 attributes [2]. Please note that finding a very large dataset is a good way to test the efficiency of the algorithm. It can also be a good idea to use a small dataset to easily test the accuracy of the algorithm.

1. Process the data and convert it into a two-dimensional array where each column represents an attribute.

This part is very simple to do when the data is purely numbers, but unfortunately, more often than not, sample data with a large number of attributes may also include qualitative attributes, such as F for woman and M for man. For this reason writing a special function that goes through each column and changes the qualitative values to numerical ones is a good idea. In the case of a column consisting of F and M it can simply be changed to zeros and ones.

1. Make an ordered list of variables.

Each variable must have an ID and it should point to a column from the dataset. Additionally the list must be sorted in ascending order by the ID number.

1. Make a correlation table from the two-dimensional array of data.

This is just one line of code using the Apache Commons library in java:

double[][] corrTable = new PearsonsCorrelation(arrayOfAllData).getCorrelationMatrix.getData();

1. Choose the thresholds T1 and T2 and create the true and false tables as the one used in the introduction’s example.

For easier demonstration of the algorithm we assume T1 = T2 so only one true/false table will be used. Let’s say the threshold is 0.7, this means that if the absolute value of an entry in the correlation matrix is greater than 0.7, then set it as TRUE, FALSE o/w.

1. Create a Function object that contains the fields for the dependent variable and the list of independent variables. It is important that this Function object **overrides the equals method** in java to avoid the addition of duplicate functions to the set.

And finally everything is ready to work on the algorithm, which consists of two parts: finding all the correlated variables with each variable from the dataset; computing all the possible sets where the correlated variables are independent from each other for a given dependent variable.

**Part I**

1. Select a dependent variable and compare it to the rest of variables, if they are correlated, add the variable to a list of correlated variables.
2. If the list of correlated Variables is not empty call the method from part II of the algorithm. o/w, add the function to the set of functions.
3. Continue step 1 and step 2 till the end of the list of variables is reached.

ComputeFunctions (**list** as ArrayList of variables, **corrTable** as Boolean table, **set** as HashSet of Functions) {

for i = 0 to i < **list**.size {

initialize corrVars as empty ArrayList

initialize y as **list**[i]

for j = 0 to j < **list**.size {

initialize x

set x to **list**[j]

if (y is not equal to x and **corrTable**[y.id][x.id] is TRUE) then {

add x to corrVars

}

}

if (corrVars is not empty) then {

initialize f as Function

set dependent variable of f to y

computeFunctionSubsets(corrVars, corrTable, f, **set**)

}

}

}

**Part II**

The function for part II is very similar to part I, except that now we are looking for the data variables to be non-correlated to each other.

1. Select a variable from the list and compare it to the rest, if they are not correlated add to a list of non-correlated variables.
2. If the list of non-correlated variables is not empty make a recursive call using the list of non-correlated variables, else add the computed function to the set.
3. Continue step 1 and step 2 till the end of the list of variables is reached.

ComputeFunctionSubsets (**list** as ArrayList of variables, **corrTable** as Boolean table, **f** as Function, **set** as HashSet of Functions) {

for i = 0 to i < **list**.size 1 {

Initialize nonCorrVars as ArrayList of variables

Initialize current as **list**[i]

for j = 0 to j < **list**.size {

Initialize candidate as list[j]

if (corrTable[current.id][candidate.id] is FALSE) then {

add candidate to nonCorrVars

}

}

Initialize updatedFunction as Function with parameters from f

add current to updatedFunction as independent variable

if (nonCorrVars is empty) then {

add updatedFunction to set

}

else {

computeFunctionSubsets (nonCorrVars, **corrTable**, updatedFunction, **set**)

}

}

}

Although this approach is guaranteed to compute all the possible functions, it will make several unnecessary computations and thus end up adding multiple duplicate functions to the Set. This is why the use of HashSets is important, since it avoids duplicates and each HashSet addition takes O(1) time.

This algorithm will work really well for most datasets, even for those of about 100 variables, yet for circumstances that involve plenty of possible multiple regression functions with several independent variables, the algorithm’s computing speed will increase exponentially.

So let’s look at an optimization, but first let’s understand better how the algorithm works, in the original computeFunctionSubsets we would take each variable from the list, compare it to all of the variables from the same list, add those that are not correlated to the nonCorrVars list and call the computeFunctionSubsets recursively. For better understanding let’s look at how the algorithm works using the following correlation matrix:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | X0 | X1 | X2 | X3 | X4 |
| X0 | T | F | F | T | T |
| X1 | F | T | F | T | F |
| X2 | F | F | T | T | T |
| X3 | T | T | T | T | F |
| X4 | T | F | T | F | T |

Fig. 2. First iteration on correlation matrix A

Ignore the first part of the algorithm and imagine this data is what we pass to the computeFunctionSubsets method. Also for simplicity sake assume that the Function is a set of independent variables. So the list is {X0, X1, X2, X3, X4}, and since we want non-correlated subsets we will only pay attention to the FALSE cells.

First iteration:

1. Select first variable in the list, which is X0
2. Compare X0 with each variable from {X0, X1, X2, X3, X4}
3. Add X1 and X2 to the list of nonCorrVars
4. Add X0 to a copy of the function, so the updated function has X0 as its independent variable.
5. Recursive call, using nonCorrVars and the updated function as parameters, yields the following matrix representation:

|  |  |  |
| --- | --- | --- |
|  | X1 | X2 |
| X1 | T | F |
| X2 | F | T |

Fig. 3. Submatrix of A after first iteration

Here we see that X1 and X2 are not correlated to each other and thus the end result of the first iteration will be just one function having {X0, X1, X2} as its independent variables list, yet before realizing this the program will attempt another iteration selecting X2 and making two extra unnecessary comparisons, it will come up with the same resulting function and will attempt to add it to the Set of functions.

Second iteration:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | X0 | X1 | X2 | X3 | X4 |
| X0 | T | F | F | T | T |
| X1 | F | T | F | T | F |
| X2 | F | F | T | T | T |
| X3 | F | T | T | T | F |
| X4 | T | F | T | F | T |

Fig. 4. Second iteration on correlation matrix A

1. Select second variable in the list, which is X1
2. Compare X1 with each variable from {X0, X1, X2, X3, X4}
3. Add X0, X2, X3 to the list of nonCorrVars
4. Add X1 to a copy of the function, so the updated function has X1 as its independent variable.
5. Recursive call using vars and the updated function as parameters yields the following matrix representation:

|  |  |  |  |
| --- | --- | --- | --- |
|  | X0 | X2 | X3 |
| X0 | T | F | T |
| X2 | F | T | T |
| X3 | T | T | T |

Fig. 5. Submatrix of A after second iteration

Sublist’s 1st iteration results: {X1 X0 X2} (Has already been added)

Sublist’s 2nd iteration results: {X1 X2 X0} (Has already been added)

Sublist’s 3rd iteration results: {X1 X3}

And so on…

As you can see the algorithm performs several unnecessary computations. So we can do the following:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | X0 | X1 | X2 | X3 | X4 |
| ~~X0~~ | ~~T~~ | ~~F~~ | ~~F~~ | ~~T~~ | ~~T~~ |
| ~~X1~~ | ~~F~~ | ~~T~~ | ~~F~~ | ~~T~~ | ~~F~~ |
| ~~X2~~ | ~~F~~ | ~~F~~ | ~~T~~ | ~~T~~ | ~~T~~ |
| X3 | ~~T~~ | T | T | T | F |
| X4 | ~~T~~ | F | T | F | T |

Fig. 6. Improved first iteration on matrix A

First improved Iteration:

1. Select first variable in the list which is X0
2. Compare X0 with each variable from {X0, X1, X2, X3, X4}
3. Add X1 and X2 to the list of nonCorrVars
4. Scratch out the rows of X0, X1 and X2 from future operations
5. Recursive call following the same approach.

Note that all of the possible comparisons were already made among X0 and the rest of the variables, thus including X0 in the rest of the iterations is pointless. A similar idea is applied to X1 and X2 with the exception that they might have some type of relation with the rest of the non-scratched out variables, X3 and X4, thus they should not be completely forgotten.

Let’s look at the matrix from the recursive call after the first iteration:

|  |  |  |
| --- | --- | --- |
|  | X1 | X2 |
| ~~X1~~ | ~~T~~ | ~~F~~ |
| ~~X2~~ | ~~F~~ | ~~T~~ |

Fig. 7. Improved first iteration on submatrix A

Here the program computes the first row, adds the result to the set and ignores the second row’s unnecessary computations and thus also prevents the attempt of a duplicate function addition to the set.

So here we can appreciate more the improvement, the algorithm keeps removing unnecessary computations on a recursive level, thus drastically reducing the computation space and time.

Note that this strategy should only be applied to the first iteration and it does not completely avoid unnecessary computations and duplicate additions to the Set. More on that later.

**Improved Algorithm Description and Pseudocode:**

1. Select pivot variable and compare it to the rest of variables
2. If a variable is non-correlated to the pivot, add it to a list
3. Add the pivot variable to the function
4. Scratch out pivot
5. Check size of list:
6. If empty, add function to set.
7. Else, scratch out variables in list and make a recursive call using the list and the new function as parameters.
8. Continue steps 1 to 5 for the rest of non-scratched out variables without scratching out any variable again.

ComputeFunctionSubsetsOptimized (**list** as ArrayList of variables, **corrTable** as Boolean correlation table, **f** as Function, **set** as HashSet of Functions) {

Initialize pivot as 0

Initialize rows as copy of **list**

Initialize pivotVar as rows[pivot]

Initialize nonCorrVars as empty list of variables

remove pivotVar from rows

for i = 0 to i < **list**.size {

Initialize candidate as **list**[i]

if (corrTable[pivotVar.id][candidate.id] is FALSE) then {

add candidate to nonCorrVars

remove candidate from rows

}

}

Initialize updatedPivotFunction as copy of **f**

add pivotVar to updatedPivotFunction as independent variable

if (nonCorrVars is empty) then {

add updatedPivotFunction to set

}

else {

ComputeFunctionSubsetsOptimized(

nonCorrVars, **corrTable**, updatedPivotFunction, **set**)

}

for i = 0 to i < rows.size {

Initialize nonCorrVars2 as empty list of variables

Initialize current as rows[i]

for j = 0 to j < list.size {

Initialize candidate as **list**[j]

if (**corrTable**[current.id][candidate.id] is FALSE) then {

add candidate to nonCorrVars2

}

}

Initialize updatedFunction as copy of **f**

add current to updatedFunction as independent variable

if (nonCorrVars2 is empty) then {

add updatedFunction to **set**

}

else {

computeFunctionSubsetsOptimized(

nonCorrVars2, **corrTable**, updatedFunction, **set**)

}

}

}

Although this algorithm will work more efficiently than the previous one, it is limited to only selecting the first row as pivot, and thus, if, let’s say, we have a matrix like the following:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | X0 | X1 | X2 | X3 | X4 |
| ~~X0~~ | ~~T~~ | ~~T~~ | ~~T~~ | ~~T~~ | ~~T~~ |
| X1 | T | F | F | F | F |
| X2 | T | F | F | F | F |
| X3 | T | F | F | F | F |
| X4 | T | F | F | F | F |

Fig. 8. Demonstration of the algorithm’s drawback

You can see the algorithm will only scratch out X0 since it’s correlation is TRUE for each other variable, and it will then perform several unnecessary computations to just return one function ({X1, X2, X3, X4}). This will still work more efficiently than the brute force algorithm from before since a better pivot selection will kick in at the recursive levels of each subsequent iteration. **But, as with quicksort, we can optimize the algorithm even further by adding a randomized pivot selection or a pivot that yields the largest list of non-correlated variables to scratch out as many unnecessary computations as possible.**

Although this new algorithm is quite elegant, it still doesn’t entirely avoid unnecessary computations and duplicate additions, thus the use of HashSets to avoid duplicates is imperative. Coming up with a similar algorithm that completely avoids the computation and addition of duplicates has some similarity with the Hamiltonian path/cycle problems, which are **NP-complete problems**. The Hamiltonian path problem consists in determining whether a path that visits each node in a graph exactly once exists, whether it is an undirected or directed graph.

EXPERIMENTAL RESULTS

Running the program for the previously mentioned dataset of 99 variables with T1 = T2 = 0.7 took 29 seconds on a home computer. Of this 29 seconds, about half the time went into reading the data and computing the matrix correlation, **0** **milliseconds** went into running the ComputeFunctions (randomized version) algorithm, and the rest went into calculating the parameters, the SSR and sorting the results according to SSR to be used for display purposes. The output consisted of 696 functions of size not greater than *p* = 3 (number of independent variables in a multiple regression function).

Note that setting different values for T1 and T2 can greatly increase or decrease the number of computed functions and size *p*. For instance, when setting T1 = 0.5 and T2 = 0.8, the number of computed functions using the same dataset becomes 9772, where the maximum number for *p* is now 10, yet the running time of the main algorithm doesn’t change much (62 milliseconds).

All the code can be found at https://github.com/AntonKovalyov1/MultipleRegression.

CONCLUSION

Although the results talk really well about the efficiency of the program’s main algorithm, in reality they will vary depending on the data, the number of variables, the distribution of relations among them and the thresholds T1 and T2 chosen. Coming up with good values for T1 and T2 is very important and can greatly reduce or increase the number of computed multiple regression functions as well as their size. Yet after experimenting with different values for T1 and T2 the main algorithm’s running speed, using the huge dataset of 99 variables previously mentioned, is affected by only a few milliseconds.

The main goal of this research is coming up with a practical and fast means of finding different relations in big datasets. We choose a huge dataset of 99 variables, simply to see how the algorithm performs in extreme situations, and it can be concluded that the algorithm’s performance did not disappoint.

REFERENCES

[1] Shmueli, Galit, Nitin R. Patel, and Peter C. Bruce. *Data mining for business intelligence: concepts, techniques, and applications in Microsoft Office Excel with XLMiner*. Hoboken: John Wiley & Sons, 2010. Print.

[2] *American Housing Survey: Housing Affordability Data System | HUD USER*. N.p., n.d. Web. 4 Apr. 2017.