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

*By Anton Kovalyov*

For this project I use a sample data set of 99 variables. The aim is to find all possible maximum size linear multiple regression functions where the dependent variable is correlated to each one of the independent variables on the right hand side of the equation, and at the same time, each one of the independent variables is not correlated to the rest of the independent variables. This can help seeing relations among different variables, and can greatly reduce the computation time space for the backward elimination algorithm.

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 |

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 everything is ready for the backward elimination algorithm to find the functions with the greatest correlation.

**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 I 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.

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

This part would be quite simple to do if all the data would be numbers, but unfortunately more often than not, sample data with a large number of attributes also consists of qualitative attributes, such as F for woman and M for man. For this reason I wrote a special method that goes through each column and changes the qualitative values to numerical ones. In the case of a column consisting of F and M it will be changed to zeros and ones.

1. Create 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. Convert the two-dimensional array of data into a correlation matrix.

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

1. Choose a threshold for defining if a variable is correlated with another and create a true and false table as the one used in the introduction’s example.

For this part I choose the value 0.7 as the threshold. 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 duplicates once we proceed working on the algorithm.

**The Algorithm**

And finally everything is ready to work on the algorithm which consists of two parts: finding all of the correlated variables with each variable from the dataset; computing all of 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, if they are correlated add to a list of correlated variables.
2. If the list of correlated Variables is not empty call the function from part

II of the algorithm.

1. 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 of the possible functions it will also make several unnecessary computations and thus end up adding multiple duplicate functions to the Set, and 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 decrease exponentially.

So let’s look at an optimization, but first let’s better understand 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 purposes let’s look at how it works using the following sample 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 |

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 is {X0}
5. Recursive call, using nonCorrVars and the updated function as parameters, yields the following matrix representation:

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

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 |

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 is {X1}
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 |

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 |

First improved Iteration:

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. Scratch 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 except for 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~~ |

Here the program computes the first row, adds the result to the set and ignores the second row’s unnecessary computations and thus duplicate addition to the set.

So here we can appreciate even 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 off variables in list and make 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 |

You can see the algorithm will only scratch 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 as many unnecessary computations as possible.**

Although this new algorithm is quite elegant it still doesn’t avoid completely unnecessary computations and additions, and thus the use of HashSets to avoid duplicates is imperative. Coming up with a similar algorithm that avoids completely 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.

**Conclusion**

Running the program for a dataset of 99 variables took 29 seconds, of which about 20 seconds went onto pre-processing (most of which went into reading the data and computing the matrix correlation), 0 **milliseconds** in running the ComputeFunctions (randomized version) algorithm, and 9 seconds in post-processing (calculating the parameters, SSR and sorting the results).

Although this results seem to talk really well about the efficiency of the main algorithm of the program, in reality they will vary depending on the data, the number of variables and the distribution of relations among them.