Max-Linear Models Code Document

Jean-Yves Djamen

Fall 2019

Contents

1 Generating ML Coefficient Matrix From a Graph

```r
ml_gen<- function(n, mat_vector){
  #We will assume that the function takes in as input n,
  #and an nxn matrix detailing the values of each edge
  #It will retrun the ML coef matrix

  #create adjacency matrix from input.
  edge_adjacency<- t(matrix(mat_vector, nrow=n, ncol=n))
  #creates directed network graph from matrix with weights as input
  network_graph<-graph_from_adjacency_matrix(edge_adjacency,
                                             mode="directed", weighted=TRUE)

  #Finds all simple paths from a starting node
  paths<-list()
  for(e in 1:n)
    paths<-c(paths,(all_simple_paths(graph=network_graph,from=e, mode="out")))
  #initialize path value list
  path_vals<-list()
  #for every path, compose coeficients
  for(p in paths){
    starting_node=p[0,1]
    first_coef=edge_adjacency[starting_node,starting_node]
    inner_prod=1
    #path coefficients (after initial one )
    for(i in 2:length(p)){
      current_coef<-edge_adjacency[p[i-1],p[i]]
      inner_prod=current_coef*inner_prod
    }
    #make list of path products
    path_vals<-c(path_vals,first_coef*inner_prod)
  }
  #matrix to be returned
  ml.coef.mat<-matrix(0,n,n)
  #fill in the diagonal entries
  for(i in 1:n){
    ml.coef.mat[i,i]=edge_adjacency[i,i]
  }
  #fill out the values at each cell
  for(e in 1:length(path_vals)){
    current.path=paths[e]
    source=current.path[[1]][1]
    destination=current.path[[1]][length(current.path[[1]])]
    #the value only gets filled out if current value is less than the desired one
    if(ml.coef.mat[source,destination]<path_vals[e]){
      ml.coef.mat[source,destination]<-path_vals[e][[1]]
    }
  }
  return(ml.coef.mat)
}
graph_gen<-function(n,mat_vector){
  #This function can be used to visualize graphs
  mat<- matrix(mat_vector, nrow=n, ncol=n)
  colnames(mat)=rownames(mat)=1:n
  network=graph_from_adjacency_matrix(t(mat))
  #network=graph_from_adj_list((mat))
  #network=graph_from_incidence_matrix(t(mat),directed=TRUE,mode="all")
  plot(network)
}
```

We will assume that the input is a graph $\mathcal{D} = (V, E)$ with n vertices. Our function `ml_gen` will take in as input an adjacency matrix A where $A_{ij} = c_{ij}$. The example below uses the graph in example 2.1 of the report.

$$A = \begin{bmatrix} c_{11} & c_{12} & c_{13} & c_{14} \\ c_{21} & c_{22} & c_{23} & c_{24} \\ c_{31} & c_{32} & c_{33} & c_{34} \\ c_{41} & c_{42} & c_{43} & c_{44} \end{bmatrix} = \begin{bmatrix} 2 & 3 & 0 & 0 \\ 0 & 2 & 3 & 4 \\ 0 & 0 & 3 & 5 \\ 0 & 0 & 0 & 2 \end{bmatrix}$$

output the ML coefficient matrix B as shown below.

```
m<- 4
m_mat<-c(2,3,0,0,
         0,2,3,4,
         0,0,3,5,
         0,0,0,2)
coef.mat<-ml_gen(m,m_mat)
#graph_gen(4,m_mat)
coef.mat
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    2    6   18   90
## [2,]    0    2    6   30
## [3,]    0    0    3   15
## [4,]    0    0    0    2
```

When doing the calculation by hand, we get the same matrix

$$B = \begin{bmatrix} 2 & 6 & 18 & 90 \\ 0 & 2 & 6 & 30 \\ 0 & 0 & 3 & 15 \\ 0 & 0 & 0 & 2 \end{bmatrix}$$

2 Generating some data

To generate the data, we need to use the coefficient matrix B. From the analysis of this matrix, we can see that any comlumn j denotes the maximum path between anode j and each of its ancestors (denoted $An(j)$). In each column, a zero in the i^{th} row means that there is no path between i and j. So, to build the function that will generate data from the coefficient matrix, we compute the element wise product of our random vector Z and each column B_i. Then, the maximum element in each of these will give us the value for our recursive structural model. In the following function this step is repeated several times to produce several sample points.

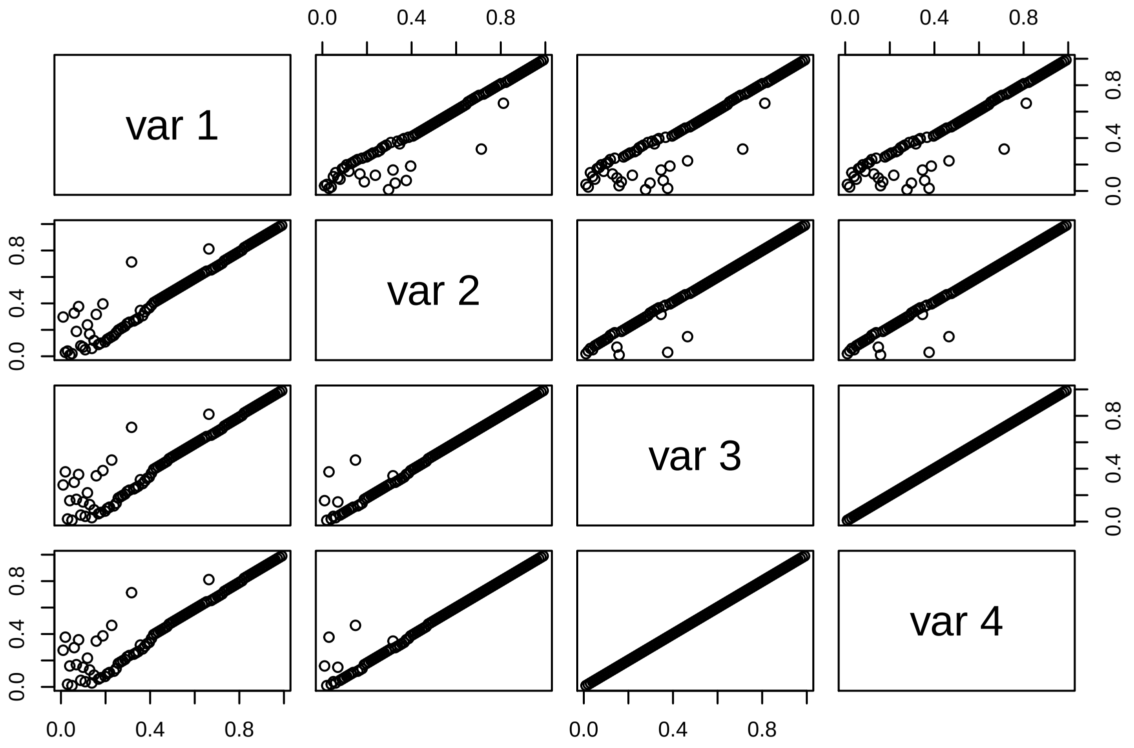
```
#given a max linear matrix and an optional input, generate some data
data_gen<-function(ml.mat, dist="exponential",lambda=1, s=1, alpha=1, m=0, samples=100){
  #lambda,s ,alpha, m are all parameters for the distribution
  #samples denotes how many samples we woiuld like to generate
  #This is the number of nodes in our graph
  n<-nrow(ml.mat)
  if(dist=="frechet"){
    #s is scale, alpha is shape and m is location
    z<-rfrechet(n*samples, loc=m, scale=s, shape=alpha)
  }

  else{
    z<-rexp(n*samples, rate=1)
  }

#turning Z into a matrix of (n x samples) size where n is the number of nodes
#in graph and samples is the number of samples we want to generate
  z=matrix(z, nrow=n, ncol=samples)

  #now we use our max prod function to calculate the max prod for each of our functions
  return(t(apply(z, 2, function(x) max_times_prd(x, ml.mat))))
}

max_times_prd<-function(mat1,mat2){
  #This function takes in two matrixes and does the max times algebta
  #here mat 1 is the Z column and mat2 is the coefficient matrix
  return(apply(mat2, 2, function(x) max(mat1*x)))
}
```

Exponential Data

With the exponential data we observe the following:

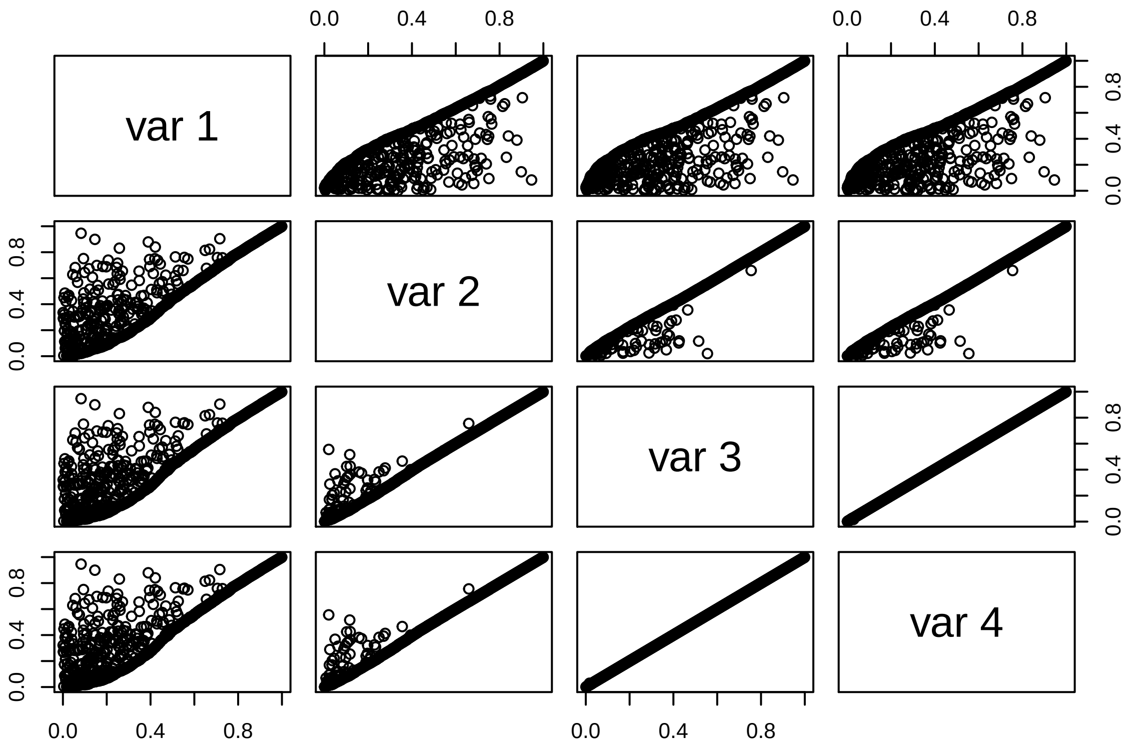
100 Samples

```
exponential.1<-data_gen(coef.mat,dist="exponential",samples=100)
pobs <- apply(exponential.1,2,rank)/(101)
pairs(pobs)
```



1000 Samples

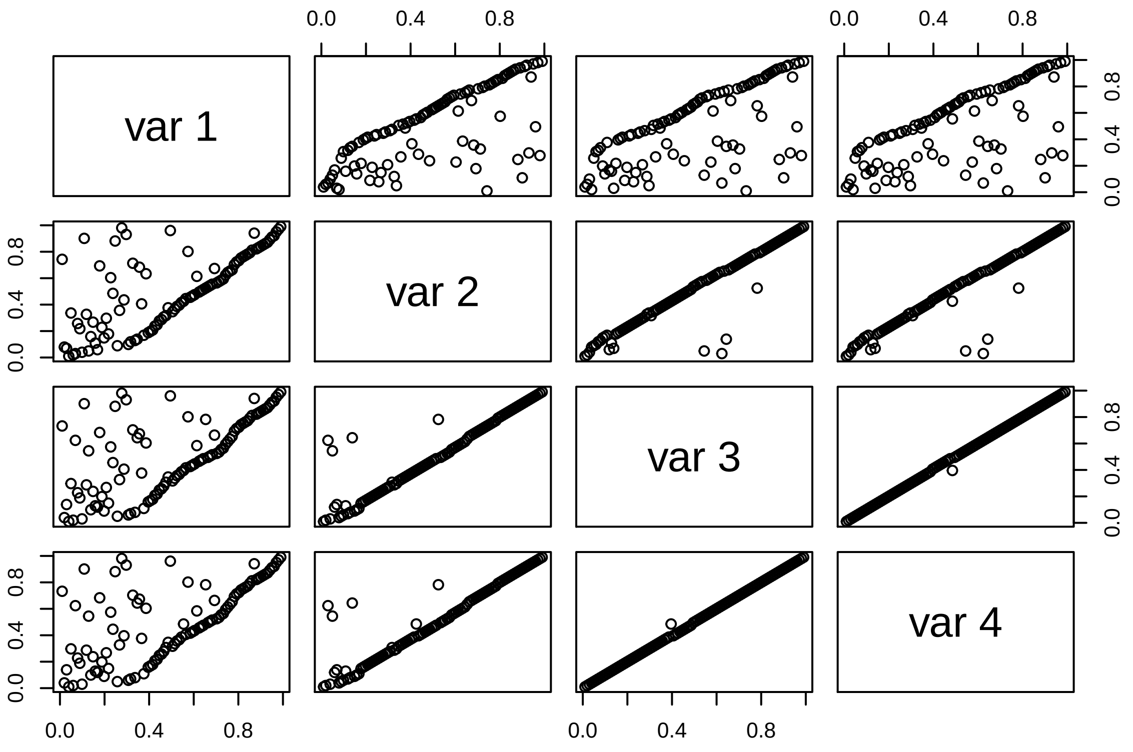
```
exponential.2<-data_gen(coef.mat,dist="exponential",samples=1000)
pobs <- apply(exponential.2,2,rank)/(1001)
pairs(pobs)
```



Frechet Distribution

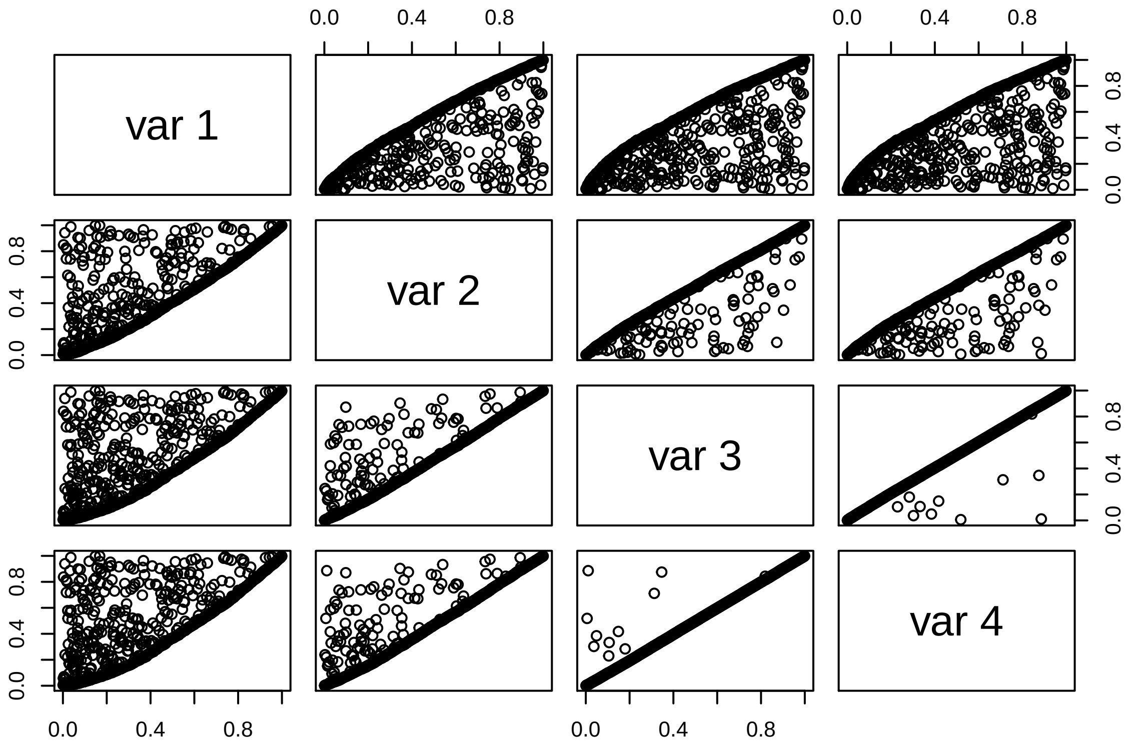
100 Samples

```
frechet.1<-data_gen(coef.mat,dist="frechet",samples=100)
pobs <- apply(frechet.1,2,rank)/(101)
pairs(pobs)
```



1000 Samples

```
frechet.2<-data_gen(coef.mat,dist="frechet",samples=1000)
pobs <- apply(frechet.2,2,rank)/(1001)
pairs(pobs)
```



3 Learning $A(t)$ from B Matrix

In this section, we will try to learn the $A_{ij}(t)$ for all the pairs X_i, X_j in our B Matrix. Unless explicitly stated, we will assume we are working with Frechét shock.

First, we write a function to compute the C matrix. The entries of this matrix are defined as $c_{ij} = \frac{b_{ij}^{\alpha}}{\sum_{k=1}^{d} b_{kj}^{\alpha}}$

```
#adjacency matrix
m_mat<-c(2,3,4,1,
         0,2,0,4,
         0,0,3,5,
         0,0,0,2)
#b matrix
coef.mat<-ml_gen(m,m_mat)
#function to calculate the sum a_ks
c_ij<-function(column,alpha){
  column_exp<-column^alpha
  #column sum
  a_j<-sum(column_exp)
  #simple return statement
  return (column_exp/a_j)
}
c_generator<-function(b_mat, alpha){
  #for each column in the matrix calculate c_ij
  return (apply(b_mat, 2, function(col) c_ij(col, alpha)))
}
#now we generate the whole matrix
c_matrix<-c_generator(coef.mat,2)
c_matrix
```

```
##      [,1] [,2]      [,3]        [,4]
## [1,]    1  0.9 0.8767123 0.845219229
## [2,]    0  0.1 0.0000000 0.033808769
## [3,]    0  0.0 0.1232877 0.118858954
## [4,]    0  0.0 0.0000000 0.002113048
```

Now that we can construct this matrix, we will write a function to learn $A(t)$ based on the C matrix. To do this, we need to consider the $d(d-1)$ unique pairs i, j of the columns of the C matrix. Each of these will have $|An(i) \cup An(j)| \leq d$ atoms to consider.

```
one_read<-function(r_1,r_2){
  #this function takes in one read of two columns
  #outputs the probability and atoms associated with the read.
  # if there is not one, it outputs none
  if(r_1==0 && r_2==0){
    #either return -1 or return null
    return(c(-1,0))
  }
  #first we calculate the probability
  p_j<-r_1/2+r_2/2
  #then we calculate the atom location
  q_j<-r_1/(2*p_j)
  #return (atom,probability)
  return (c(q_j,p_j))
}
#computes the atoms and probabilities associated with two columns of our c matrix
paired_prob_atom_generator<-function(row_1,row_2){
  #this function takes in two rows of the C matrix
  #outputs the atoms and probabilities associated with this pairing
  return(rbind(mapply(function(r_1,r_2) one_read(r_1,r_2),row_1,row_2)))
}

#this function takes in the c matrix and outputs the density points
prob_atom_generator<-function(c_matrix){
  n<-ncol(c_matrix)
  #this loop just gives me all distinct pairings of the rvs in the c matrix
  distinct_pairs<-c()

  #if we want to compare d(d-1) interactions
  for(i in 1:n){
    for(j in i+1:n){
      if(j<=n) distinct_pairs<-rbind(distinct_pairs,c(i,j))
    }
  }

  #return(distinct_pairs)
  output<-data.frame()
  #now distinct_pairs is the distinct pairs
  #we can iterate through this list to ps and qs
  pqs<-apply(distinct_pairs, 1, function(x,...)
    list(name=paste(x[1],x[2],sep="_"),value=paired_prob_atom_generator(c_matrix[,x[1]],c_matrix[,x[2]]
  #now, we have a list of pq pairs.
  #each labeled with the index of the columns being compared
  return(pqs)
}
```

```r
#for any c matrix, this generates the list of pairwise ps and qs (d*(d-1)) dimensional
x<-prob_atom_generator(c_matrix)
```

Now, given a C matrix, we are able to calulate the pairwise atom locations and their associated probabilities. For completeness, the next block of code transoforms this information and calculates the continuous fucntion $A(t)$.

```r
test<-x[[1]]$value
a_t_generator<-function(pqs){
  #given a list of matrices of the form
  #qis
  #pis
  #this function outputs the function A(t)
  #unzip the list of matrixes into a list of tuples
  ps=pqs[2,]
  qs=pqs[1,]

  #transofrming into a data frame
  pqs<- data.frame("Probabilities" = ps, "Atoms" = qs)

  #sorting them by the atom location
  pqs<-pqs[order(pqs$Atoms),]
  #now we define the function that will return A(t)
  A<-function(t){
    #kinks
    atoms<- 1-pqs$Atoms
    kinks<- data.frame("Probabilities" = pqs$Probabilities,
                       "Atoms" = pqs$Atoms,"NegAtoms"=atoms)

    #add t to our kinks
    atoms["New"]<-t
    atoms2<-sort(atoms,decreasing=FALSE,index.return=TRUE)
    #see where it lies in list (first time we see a number == t)
    pos<-(which(unlist(atoms2)==t))[1]

    #now we have our position calculator
    position_calc<-function(x){
      if ((which(unlist(atoms2)==x[["NegAtoms"]]))[1] <pos ){
        return (2* (1-x[["Atoms"]]) *(1-t) * x[["Probabilities"]] )
      }
      else{
        return (2*((x[["Atoms"]])*(t))*x[["Probabilities"]])
      }
      }
    vec<-apply(kinks,1,position_calc)
    return (sum(vec))
  }

  return(A)

}
A<-a_t_generator(test)
```