Chapter_7

Henry Bourne

2022-11-28

Matrices

In this chapter we will go over various ways matrices are implemented and can be used in R.

Dense Matrices

First we will go over using dense matrices in R. A **dense matrix** is a two-dimensional data structure where we can assign entries values by column, or by row if we set *byrow=TRUE*. We can also change the name of the rows and columns. If we access a coumn/row of a matrix we obtain a vector, we can change this so instead we get a column by setting *drop=FALSE* when indexing. R also has higher dimensional data structures known as arrays, note that the matrix is just a special case of an array.

We can solve **linear systems** in R very quickly and easily using the **solve()** command. When just given a matrix, eg. *solve(A)*, it will compute the inverse and when given a matrix and a vector, eg. *solve(A,b)*, it will solve the linear equation $Ax=b$ for x. Note that when we do want to solve a linear system it is advised to use the second option as opposed to finding *solve(A)* and then multiplying b. This is because it is faster and more accurate, unless a linear system needs to be solved repeatedly for different vectors.

Numerical stability, finite precision arithmetic

Now we will briefly talk about numerical stability and finite precision arithmetic which is useful to know about when working with dense matrices.

In R a floating point number is stored as a double precision number. According to the IEEE754 standard, a double number consists of 64 binary bits arranged as follows: sign bit = 1 bit, exponent = 11 bits, significant precision = 53 bits. This means there exist largest and smallest finite numbers in R. It also means we have **finite precision**, as we can only represent a number to a precision of $2^{-53} \approx 1.11 \times 10^{-16}$, so we can expect errors of order 10^{-16} in numerical representations (and therefore computation) of double numbers.

This can not only be a problem in terms of accuracy but also when trying to compare values. For example:

```
0.1 + 0.2 - 0.3 == 0
```

```
## [1] FALSE
```

The statement above evaluates as *FALSE* even though we know this statement to be true, this is down to finite precision. To remedy this we can use **all.equal()** which ignores differences up to a tolerance level of 1.5×10^{-8} (as default):

```
all.equal(0.1+0.2-0.3, 0)
```

```
## [1] TRUE
```

Due to finite precision errors many errors can arise when using matrices, especially in matrix multiplication due to the many operations involved. For example earlier we talked about the *solve()* function and how when we want to solve linear systems it is better off letting the function handle it all due to problems with accuracy.

Sparse Matrices

The R **Matrix** package provides additional functions for both dense and sparse matrices that extend the basic matrix data type. Some usefull functions for dense matrices from this package are: **rankMatrix()** which returns the rank of the input matrix (up to a certain tolerance if desired) and **rcond()** which gives the condition number of the square matrix. Where the **condition number** is the product of the norm of the matrix and the norm of its inverse. In the *Matrix* package dense matrices are stored as **dgeMatrix** objects.

By default a sparse matrix is stored as a **dgCMatrix**, **sparse matrices** are matrices where most entries will be 0, we can use the fact a matrix is sparse to save on memory and compute. The "d" stands for digit, "g" for general (ie. not triangular or symmetric) and "C" for column (ie. not row or triplet). The benefit of storing a matrix as a *dgCMatrix* is that it is stored using compressed sparse column format (**CSC format**), alternatives are compressed sparse row format (**CSR format**) which is **dgRMatrix** or triplets which is **dgTMatrix**. The benefits of these objects are that they take up less memory than a *dgeMatrix*. Note that addition or subtraction with a sparse matrix object will result in a dense matrix and that multiplication of a sparse matrix by a dense matrix will result in a dense matrix. Also note that just like we had with dense matrices one should avoid inverting sparse matrices as they are not guaranteed to be sparse, so when trying to solve linear equations one should aim to use *solve(A,b)*.

Dependency Graphs

A graph G=(V,E) can be represented using an adjacency matrix A, where A is a square matrix with $A_{i,j} = 1$ if E contains the edge (i,j) and 0 otherwise. Note that this adjacency matrix will likely be sparse. Here we will look at an example of building and using an sparse adjacency matrix from a dependency graph.

The senate is a legislative branch of the US government where senators are (usually) affiliated to either the democratic or republican party and there are 100 senators - 2 per state. We will be looking at a dataset which contains rollcall votes (senators say "yay" or "nay" when names are called for their verdict on the bill) of the 109th US senate. Our aim will be to analyze the dependencies between senators using rollcall data. First we will load in the data:

```
# We first download the data from:
↪  https://github.com/anewgithubname/MATHM0041-2022/blob/main/lecs/senate109.zip
votes <- read.csv("senate109/votes.csv")
party <- read.csv("senate109/party.csv")
name <- read.csv("senate109/name.csv")

# We remove data pertaining to the independent party member
votes <- t(votes[-90,])
party <- ifelse(party[-90,] == -1, 0, 1)
```

We would like to model the factorization of $p(x^{(1)} \cdots x^{(100)})$, where $x^{(i)}$ is the vote senator i casts on a bill. Let's assume p is a pairwise markov network, then we have $p(x^{(1)} \cdots x^{(100)}) \propto \prod_{c \in C} g_c(x^{(c)})$ where we can further factorize each clique as $g_c = \prod_{(u,v) \in c} g_{u,v}(x^{(u)}, x^{(v)})$. Which means we can write $p(x^{(1)} \cdots x^{(100)}) \propto \prod_{(u,v) \in E} g'_{u,v}(x^{(u)}, x^{(v)})$ for some unknown function g'. If we model $g'(u,v) := \exp(w_{u,v} x^{(u)} x^{(v)})$ for all u,v where we have that if $w_{u,v} = 0$ there is no edge between nodes u and v in the graph.

Let us first model $g'_{1,v}$ (The parameter belonging to the senator named SESSIONS) to do this we would like to maximize the likelihood wrt. $w_{1,v,\forall v}$ (note what we have here is a binary logistic regression):

```
devtools::install_github("h-aze/compass_yr1", subdir = "/labs/stattools")
```

```
## Skipping install of 'stattools' from a github remote, the SHA1 (b57bb146) has not changed since last
##   Use `force = TRUE` to force installation
```

```
library(stattools)
y <- ifelse(votes[,1] == -1, 0, 1)
```

```
init_params <- rnorm(ncol(votes)+1)
SESSIONS <- optim(par = init_params, fn = binlr_nll, D=votes, y=y, method = "SANN") ;
↪  SESSIONS
```

```
## $par
##   [1] -3.15019069 18.24545823  0.78973067 -0.71534811 -2.43628968 -0.48011227
##   [7]  1.08200276  0.77347691 -1.22335571  1.08465692  0.02915985  2.53711958
##  [13]  2.71479205  0.19577138  0.44737390 -1.75840611 -1.23366844 -2.05228405
##  [19]  0.80878445  0.45841191 -1.57789641 -3.07720110  1.17108425  3.05359384
##  [25] -0.87265910 -1.47569186 -1.38065973 -3.05088933  3.02803289 -2.37392355
##  [31]  1.87516580  0.82718148  1.71328543  2.08453933  1.36834966 -1.26101801
##  [37]  1.69327736 -5.34283895  3.59753645 -2.27322512 -0.36455310  3.54189053
##  [43] -0.46563492  4.01115219 -2.09416303 -1.11584886  3.00853541  5.40651277
##  [49] -2.93154436 -2.22932431 -5.56481310 -1.58713315 -0.38288103 -4.21302585
##  [55]  0.46839339 -0.37717245  0.94110387 -1.50436760  0.37737296 -4.35674136
##  [61] -2.87160043 -1.21008900 -3.43885962  0.78601008  1.13739133  1.00167587
##  [67]  2.06592145  4.19938600 -2.68082390  3.15466702  1.92731706  1.08184890
##  [73]  1.70256013  1.39493810  0.06386125 -0.51203730 -3.52743020  2.28432195
##  [79] -1.83620646  4.14106268  2.54914099  0.79401976  0.60609741 -2.07034205
##  [85] -1.97973307  0.16593213  2.83346064 -3.90795534 -3.35396541  3.05015803
##  [91]  2.23940471  0.32699668 -1.23675733  2.11621274  0.34132232  0.69756395
##  [97] -1.57688089 -1.00124652  3.00758526  0.70000519 -0.13022624
##
## $value
## [1] 0.8255037
##
## $counts
## function gradient
##    10000       NA
##
## $convergence
## [1] 0
##
## $message
## NULL
```

We note, however, that most of the parameters here are not close to zero, to make this the case we need to add a regularization term, we will use the l1 regularization:

```
lambda <- 200000
binlr_nll.l1 <- function(par, D, y){
  binlr_nll(par,D,y) + lambda * sum(abs(par))
}
```

Now lets again try and model $g'_{1,v}$:

```
SESSIONS <- optim(par = init_params, fn = binlr_nll.l1, D=votes, y=y, method = "SANN") ;
↪  SESSIONS
```

```
## $par
##   [1]  0.9922001739 -0.0142263951 -0.2088046927  0.0559852967  0.0594108695
##   [6]  0.3124321722 -0.1293408562 -0.1920889321 -0.0662331999  1.2110086415
##  [11]  0.0297344114 -0.2147699972 -0.1677221167  0.0003152048 -0.1067891938
##  [16]  0.2312016419 -0.3685069015 -0.4315382988 -0.6549604886  0.2979738247
##  [21] -0.0311689039 -0.1981469366  0.3100627263 -0.1001064374 -0.1496494856
```

```
##  [26] -0.1021189437 -0.0746295591 -0.1530724752  0.0324196019 -0.0367820159
##  [31] -0.1709587624 -0.5715930771 -0.1197380604  0.2582785053  0.0155127223
##  [36] -0.1812857789 -0.0113723874 -0.0164924107 -0.0467357315  0.0422992733
##  [41] -0.1955186205  0.1105252807 -0.0880693563  0.2042863419  0.7910293773
##  [46] -0.2995086232  0.0021510883 -0.1341885450 -0.1348006539  0.0014144318
##  [51]  0.0564519525  0.2429877399  0.0520780203  0.0733070440  0.0507309558
##  [56] -0.2463867112 -0.1939154633 -0.1032434787 -0.2777614741  0.0181794692
##  [61] -0.1720590925 -0.3272807618  1.5858957983 -0.0764048887  0.0887102999
##  [66] -0.0725922485 -0.0592123312  0.0969977813  0.5461636477  0.0539753016
##  [71]  0.1044817806 -0.0332543314 -0.2940327159  0.0941494684 -0.0540829127
##  [76]  0.0327456659 -0.0286755548 -0.2328910947 -0.0173383273  0.1215845625
##  [81] -0.4051775174  0.0368866463  0.2103784080 -0.2323920586  0.1663705169
##  [86] -0.1414428684 -0.1723874262 -0.0628330135 -0.0859546256  0.0173855364
##  [91]  0.1467269598 -0.0111798951  0.0252095940 -0.0363966628 -0.0914332890
##  [96] -0.5491884679  0.1713012326  0.0627507601  0.0749814379  0.1196291326
## [101] -0.1448358732
##
## $value
## [1] 3742209
##
## $counts
## function gradient
##    10000       NA
##
## $convergence
## [1] 0
##
## $message
## NULL
```

Here alot more of the parameters are close to zero. We can now map values that are close to zero to zero:

```
SESSIONS_tidy <- ifelse(SESSIONS$par < 0.5, 0, SESSIONS$par) ; SESSIONS_tidy
```

```
##   [1] 0.9922002 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000
##   [8] 0.0000000 0.0000000 1.2110086 0.0000000 0.0000000 0.0000000 0.0000000
##  [15] 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000
##  [22] 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000
##  [29] 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000
##  [36] 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000
##  [43] 0.0000000 0.0000000 0.7910294 0.0000000 0.0000000 0.0000000 0.0000000
##  [50] 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000
##  [57] 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000 1.5858958
##  [64] 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000 0.5461636 0.0000000
##  [71] 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000
##  [78] 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000
##  [85] 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000
##  [92] 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000
##  [99] 0.0000000 0.0000000 0.0000000
```

Now we see that most of our values are now zero and from this we can infer which senators SESSIONS tends to vote the same as.

Now we would like to do this for all the senators and store the results in a spare matrix to reduce memory, where this sparse matrix is the adjacency matrix for the graph:

```
library(Matrix)
connections <- c()
for(j in 1:ncol(votes)){
  y <- ifelse(votes[,j] == -1, 0, 1)
  senator_j <- optim(par = init_params, fn = binlr_nll, D=votes, y=y , method = "SANN")
  connected_to <- c()
  for(i in 2:length(senator_j$par)){
    if(senator_j$par[i] > 0.5){
      connected_to <- c(connected_to,1)
    }
    else{
      connected_to <- c(connected_to,0)
    }
  }
  connections <- c(connections, connected_to)
}
g <- Matrix(connections, nrow=ncol(votes), ncol=ncol(votes), sparse=TRUE)
```

We can then plot the graph we get using our sparse adjacency matrix:

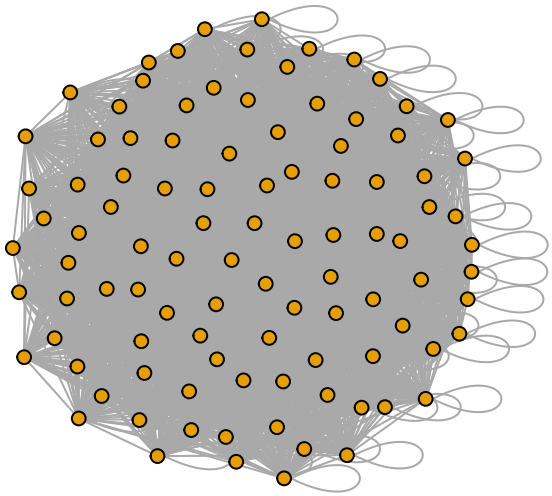
```
library(igraph)
```

```
##
## Attaching package: 'igraph'
```

```
## The following objects are masked from 'package:stats':
##
##     decompose, spectrum
```

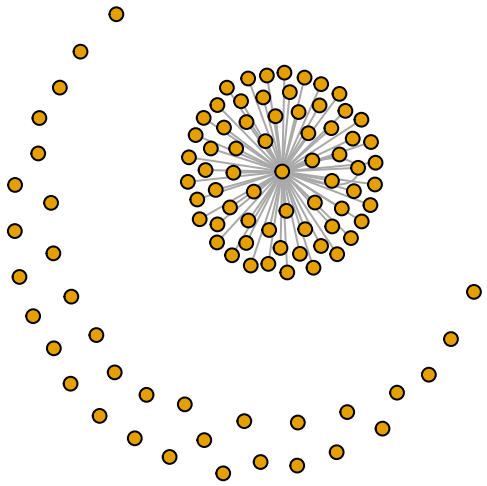
```
## The following object is masked from 'package:base':
##
##     union
```

```
G <- graph.adjacency(g)
G<- as.undirected(G)
plot(G, vertex.size=6, vertex.label=NA)
```



Note that it is very hard to tell anything from this graph as there are too many nodes and connections. Let's instead analyse the connections of one node at a time, for example let's just plot the graph with edges included if it's incident on node 26 which belongs to OBAMA:

```
gdash <-  Matrix(0, nrow = ncol(votes), ncol = ncol(votes), sparse=TRUE)
gdash[,26] <- g[,26]
gdash[26,] <- g[26,]
G <- graph.adjacency(gdash)
G<- as.undirected(G)
plot(G, vertex.size=6, vertex.label=NA)
```



We see here that OBAMA votes similarly to some of the other senators, but not all, specifically he votes similarly to:

```
agrees <- c()
for(i in 1:nrow(g)){
  if(g[i,1] == 1){
    agrees <- c(agrees, name[i,])
  }
}
agrees
```

```
##  [1] "SESSIONS"    "FEINSTEIN"   "CHAMBLISS"   "AKAKA"       "INOUYE"
##  [6] "CRAIG"       "GRASSLEY"    "BROWNBACK"   "ROBERTS"     "BUNNING"
## [11] "MCCONNELL"   "VITTER"      "COLLINS"     "SARBANES"    "KERRY  JOHN"
## [16] "STABENOW"    "COLEMAN"     "COCHRAN"     "BAUCUS"      "NELSON  BEN"
## [21] "LAUTENBERG"  "BINGAMAN"    "DOMENICI"    "SCHUMER"     "DORGAN"
## [26] "DEWINE"      "VOINOVICH"   "INHOFE"      "COBURN"      "SMITH  GORD"
## [31] "CHAFEE"      "DEMINT"      "GRAHAM"      "THUNE"       "JOHNSON"
## [36] "CORNYN"      "HATCH"       "JEFFORDS"    "LEAHY"       "ALLEN"
## [41] "WARNER"      "CANTWELL"    "BYRD  ROBER" "FEINGOLD"    "KOHL"
```

And he votes disimilarly to:

```
disagrees <- c()
for(i in 1:nrow(g)){
  if(g[i,1] == 0){
    disagrees <- c(disagrees, name[i,])
  }
```

```
}
disagrees
```

```
##  [1] "SHELBY"      "MURKOWSKI"   "STEVENS"     "KYL"         "MCCAIN"
##  [6] "PRYOR"       "LINCOLN"     "BOXER"       "ALLARD"      "SALAZAR"
## [11] "DODD"        "LIEBERMAN"   "BIDEN"       "CARPER"      "MARTINEZ"
## [16] "NELSON"      "ISAKSON"     "CRAPO"       "DURBIN"      "OBAMA"
## [21] "BAYH"        "LUGAR"       "HARKIN"      "LANDRIEU"    "SNOWE"
## [26] "MIKULSKI"    "KENNEDY  ED" "LEVIN  CARL" "DAYTON"      "LOTT"
## [31] "TALENT"      "BOND"        "BURNS"       "HAGEL"       "ENSIGN"
## [36] "REID"        "GREGG"       "SUNUNU"      "CORZINE"     "MENENDEZ"
## [41] "CLINTON"     "BURR"        "DOLE"        "CONRAD"      "WYDEN"
## [46] "SANTORUM"    "SPECTER"     "REED"        "FRIST"       "ALEXANDER"
## [51] "HUTCHISON"   "BENNETT"     "MURRAY"      "ROCKEFELLER" "ENZI"
```