Converting data formats: Making a nexus file from csv data

Creating a nexus file

We'd like to make a nexus file from some linguistic data, so we can use it in programs like Mesquite and Splitstree.

The easiest way to see the structure of a nexus file is to look at an example. You can open a nexus file in a good text editor. Here's a very simple nexus file for 5 languages, storing two variables of binary data.

```
#NEXUS

BEGIN TAXA;
    TITLE Taxa;
    DIMENSIONS NTAX=5;
    TAXLABELS
        Welsh
        Irish
        Breton
        Cornish
        Manx
    ;
END;



BEGIN CHARACTERS;
    TITLE  Character_Matrix;
    LINK TAXA = Taxa;
    DIMENSIONS  NCHAR=2;
    FORMAT DATATYPE = STANDARD GAP = - MISSING = ? SYMBOLS = "  0 1";
    MATRIX
    Welsh  1 1
    Irish  1 0
    Breton  0 1
    Cornish  1 1
    Manx  0 0

;
```

This data has:

- A '#NEXUS' line at the start, to define the file type
- A 'TAXA' block, which lists the languages. Note that there's a line with 'NTAX' and a number - this defines how many languages the file should hold.
- A 'CHARACTERS' block, which lists the data for the langauges. It's has some information about the format of the data, then a 'MATRIX' section, which has a list of the languages, then a row of 1s and 0s, defining the traits of the two variables.
- Other fiddly bits and pieces which look important.

We'd like to load some data in R, then save it to a file format like this. For the csv format, we could just use the 'write.csv' function, but there's no no built-in function to make a nexus file. We can make our own!

Preparing data

First, let's load the data from WALS, loaded from a csv file.

Add this to a new script file:

```
setwd("~/Documents/Teaching/JenaSpringSchool/org/spring-school/IntroToR")

d <- read.csv("data/WALS_WordOrder.csv", stringsAsFactors = F)
glottoData <- read.csv("data/Glottolog_Data.csv", stringsAsFactors = F)
d2 <- merge(d, glottoData, by.x='glottocode', by.y='glotto.code')
```

And let's extract from that just the data on languages from the Atlantic Congo family. To make things simple, we'll just look at langauges which have either SVO or SOV and either Prepostions or Postpositions (so the variables are binary).

```
# Load data for Atlantic Congo
data = d2[d2$glotto.family=="Atlantic-Congo",]
# use only binary categories
data = data[data$BasicWordOrder %in% c("SVO","SOV"), ]
data = data[data$AdpositionOrder %in% c("Prepositions","Postpositions"), ]
```

Calculating specific parts

In the nexus file format, some of the file format is **standard** - we need the first '#NEXUS' line in any file. And some of it is **specific** to the data - like the value of NTAX, and the list of languages. We can just copy the standard parts, but we need to calculate the specific parts. What's standard and what's specific?

The specific parts are:

- NTAX: The number of taxa (languages)
- NCHAR: The number of characters (variables)
- The list of taxa names
- The character data

We'll use the iso code to identify each language, and we'll the character data is just the two linguistic variables.

```
taxlabels = data$iso_code
characterData = data[,c("BasicWordOrder","AdpositionOrder")]
```

We can now work out the values of NTAX and NCHAR:

```
numTax = nrow(characterData)

numChar = ncol(characterData)
```

So we have our first bits! Eventually, we'll paste these into the right place inside a template. But we need the other bits first.

Character data, converting to binary

Let's take a look at the character data:

```
head(characterData)
```

```
##    BasicWordOrder AdpositionOrder
## 10            SVO    Prepositions
## 12            SVO   Postpositions
## 53            SVO    Prepositions
## 63            SVO    Prepositions
## 64            SVO   Postpositions
## 66            SVO    Prepositions
```

This isn't quite right - the data should be binary. We need a way of converting this into 1s and 0s.

One way to do this is to convert the values into a factor, then convert the factor into a number. Let's try this out with a little experiment:

```
x = c("Cake","Pie","Pie","Cake")
x
```

```
## [1] "Cake" "Pie"  "Pie"  "Cake"
```

```
factor(x)
```

```
## [1] Cake Pie  Pie  Cake
## Levels: Cake Pie
```

```
as.numeric(factor(x))
```

```
## [1] 1 2 2 1
```

This isn't quite right either - it's a list of 1s and 2s, so we need to subtract 1.

```
as.numeric(factor(x)) - 1
```

```
## [1] 0 1 1 0
```

This is a useful tool, so let's make our own custom function to convert data to binary:

```
convertToBinary = function(x){
  return(as.numeric(as.factor(x))-1)
}
```

Let's put that at the top of our script file, and run it so that it's stored in the memory. Now we can try it out on a set of our data:

```r
head(convertToBinary(characterData$AdpositionOrder))
```

```
## [1] 1 0 1 1 0 1
```

That works!

Applying a function to all columns

Now we need to convert every column in *characterData* into a binary format. We could do this with a `for` loop:

```r
for(i in 1:ncol(characterData)){
  characterData[,i] <-  convertToBinary(characterData[,i])
}
```

This can be read as "For each number in the list from 1 to the number of columns (in this case: 1, 2), do the following: Replace column i in characterData with the binary conversion of column i."

However, using for loops for lots of data can take a long time. A more 'R' approach is to use `apply`.

The function `apply` takes a matrix or data frame and applies a function to either each row or each column. Here's a toy example of how it works. Let's make a matrix with some numbers in:

```r
# Make a matrix with some numbers in
x = table(characterData$BasicWordOrder, characterData$AdpositionOrder)
x
```

```
##
##       Postpositions Prepositions
##   SOV             2            0
##   SVO            11           93
```

`apply` takes three arguments: The data, a function (like `sum`) and a number representing whether to apply the function to the rows or the columns:

```r
# take the sum of each row:
apply(x, 1, sum)
```

```
## SOV SVO
##   2 104
```

```r
# take the sum of each column:
apply(x, 2, sum)
```

```
## Postpositions  Prepositions
##            13            93
```

So, we could convert each column in *characterData* to binary using this code:

```r
characterData.binary = apply(characterData, 2, convertToBinary)
```

Let's check it worked:

```r
head(characterData.binary)
```

```
##      BasicWordOrder AdpositionOrder
## [1,]              1               1
## [2,]              1               0
## [3,]              1               1
## [4,]              1               1
## [5,]              1               0
## [6,]              1               1
```

This code will now work for 2 columns, or 200! It will also come in handy for the next step.

Making a list of taxa: using `paste`

Let's look at the TAXLABELS section of the nexus file again:

```
TAXLABELS
    Welsh
    Irish
    Breton
    Cornish
    Manx
;
```

We need a string with each language name on a different line. Recall that the language names in our WALS data are iso codes:

```r
head(taxlabels)
```

```
## [1] "fub" "adj" "asa" "bfd" "bsp" "bkc"
```

The function `paste` combines a vector of strings into a single string. It has optional arguments "sep" and "collapse" which have different effects. For example:

```r
# Combine a list of three names with a surname:
paste(c("Johnny","Joey","Tommy"), "Ramone", sep=" ")
```

```
## [1] "Johnny Ramone" "Joey Ramone"   "Tommy Ramone"
```

```r
# Join items in two lists with a hyphen:
paste(c("a","b","c"),c("A","B","C"), sep="-")
```

```
## [1] "a-A" "b-B" "c-C"
```

```r
# Combine four strings into a single string:
paste(c("super","cali","fragilistic","expialidocious"), collapse = "")
```

```
## [1] "supercalifragilisticexpialidocious"
```

We can use `paste` to create a single string that combines all the language names by new lines. New lines can be represented using the 'escape' character backslash ("\n"):

```r
taxlabels.string <- paste(taxlabels,collapse="\n")
```

The character block: Combining `apply` and `paste`:

Let's look at the format of the character block:

```
MATRIX
Welsh  1 1
Irish  1 0
Breton  0 1
Cornish  1 1
Manx  0 0

;
```

Each line starts with the taxon name, then the list of characters comes. We can make a data frame that looks like this by combining the *taxlabels* variable with the *characterData* data frame, using `cbind` (which combines things by column):

```r
characterData.binary.withNames = cbind(taxlabels,characterData.binary)

head(characterData.binary.withNames)
```

```
##      taxlabels BasicWordOrder AdpositionOrder
## [1,] "fub"     "1"            "1"
## [2,] "adj"     "1"            "0"
## [3,] "asa"     "1"            "1"
## [4,] "bfd"     "1"            "1"
## [5,] "bsp"     "1"            "0"
## [6,] "bkc"     "1"            "1"
```

Now we need to do two things:

- Paste the strings in each row into a single string, seperated by a space,
- Paste these strings into one string, seperated by a new line.

> **TASK**: There are many ways to do this, but see if you can figure out how to do this using `apply` and paste.

Here's my solution:

```
combineRows = apply(characterData.binary.withNames,
      1, paste, collapse=' ')
characterData.binary.string =
  paste(combineRows,collapse='\n')
```

Putting it all together

We now have all the bits we need to make the file:

- NTAX: The number of taxa (*numTax*)
- NCHAR: The number of characters (*numChar*)
- The list of taxa names (*taxlabels*)
- The character data (*characterData.binary.string*)

There are many ways we could put all the bits together. For example, we could make each line in the file using `paste`, then paste all these lines together.

But here's a more elegant solution: We'll use a template string with all the standard parts, with markers where we want to add the specific information.

We can define a template string by copying the text from the example file, putting it between single quotes and replacing the specific parts with unique strings we can refer to later:

(note that definitions of variables can be spread over multiple lines)

```
nexus_template =
'#NEXUS

  BEGIN TAXA;
  TITLE Taxa;
  DIMENSIONS NTAX=ntax_here;
  TAXLABELS
  taxlabels_here
  ;
  END;



  BEGIN CHARACTERS;
  TITLE  Character_Matrix;
  LINK TAXA = Taxa;
  DIMENSIONS  NCHAR=nchar_here;
  FORMAT DATATYPE = STANDARD GAP = - MISSING = ? SYMBOLS = "  0 1";
  MATRIX
  characterData_here

  ;
'
```

Now all we need to do is replace "ntax_here" with the variable *numTax* etc. To do this we can use the function `gsub` which substitutes (or "replaces""") one string with another:

```r
# change 's' to 'k' in the string 'snow':
gsub("s", "k", "snow")
```

```
## [1] "know"
```

So, we just need to copy the template text, then replace the placeholders with the bits we calculated.

```r
  # copy template to new variable
  nex = nexus_template
  # add data to template
  nex = gsub("ntax_here", numTax, nex)
  nex = gsub("nchar_here", numChar, nex)
  nex = gsub("taxlabels_here", taxlabels.string, nex)
  nex = gsub("characterData_here", characterData.binary.string, nex)
```

The final step is to write this to a file. We can use the function `cat`:

```r
  filename = "data/Atlantic-Congo.nex"
  cat(nex, file=filename)
```

And we're done! This creates a file that can be opened in programs like Mesquite (see below for the full script as it would appear in a script file).

Taking it further: Making the code re-usable

We wrote a script to convert data from Atlantic-Congo languages to a nexus file. If we want to do this for another set of data, we could just copy the whole script and replace bits and pieces. However, a much more efficient approach would be to make a custom function that takes a set of data and produces a nexus file.

> **TASK**: Create a custom function named `makeNexusFile` which takes three arguments: a list of language names (taxlabels), a data frame of character data (characterData) and a filename. It should create a nexus file format and write it to a file.

We already have all the lines of code we need, we just need to put the right ones inside a function. Have a think: what bits would need to be specified each time, and which bits can be calcualted automatically? For example, we can calculate the number of taxa directly from the character data.

Here's how the script file should be organised:

- Set the working directory
- Parameters (if we have any)
- Support functions (like the `convertToBinary` function)
- Main conversion function (`makeNexusFile`)
- Lines to load the data, make the specific data
- Call the `makeNexusFile` function

Here's my code. My function takes three variables: the list of languages, the character data and the name of the file to write to.

```r
setwd("~/Documents/Teaching/JenaSpringSchool/org/spring-school/IntroToR")

############
# PARAMETERS

# Nexus file template

  nexus_template =
    '#NEXUS

  BEGIN TAXA;
  TITLE Taxa;
  DIMENSIONS NTAX=ntax_here;
  TAXLABELS
  taxlabels_here
  ;
  END;



  BEGIN CHARACTERS;
  TITLE  Character_Matrix;
  LINK TAXA = Taxa;
  DIMENSIONS  NCHAR=nchar_here;
  FORMAT DATATYPE = STANDARD GAP = - MISSING = ? SYMBOLS = "  0 1";
  MATRIX
  characterData_here

  ;
  '


############
# FUNCTIONS

# Function to convert a vector to a binary vector
convertToBinary = function(x){
  return(as.numeric(as.factor(x))-1)
}

# Function to convert data to a nexus format
makeNexusFile = function(taxlabels, characterData, filename){

  # make tax labels string
  taxlabels.string = paste(taxlabels,collapse='\n')

  # Get number of taxa
  numTax = nrow(characterData)
  # Get number of characters (variables)
  numChar = ncol(characterData)

  # convert characters to binary
  characterData.binary = apply(characterData, 2, convertToBinary)
  # add taxon labels to character section
  characterData.binary.withNames = cbind(taxlabels,characterData.binary)

  # paste everything together
  combineRows = apply(characterData.binary.withNames,
        1, paste, collapse=' ')
  characterData.binary.string =
    paste(combineRows,collapse='\n')


  # Copy the template
  nex = nexus_template
  # add data to template by replacing markers
  nex = gsub("ntax_here", numTax, nex)
  nex = gsub("nchar_here", numChar, nex)
  nex = gsub("taxlabels_here", taxlabels.string, nex)
  nex = gsub("characterData_here", characterData.binary.string, nex)

  cat(nex, file=filename)
}

#############
# DATA

# Load the WALS data
d <- read.csv("data/WALS_WordOrder.csv", stringsAsFactors = F)
glottoData <- read.csv("data/Glottolog_Data.csv", stringsAsFactors = F)
d2 <- merge(d, glottoData, by.x='glottocode', by.y='glotto.code')

# Load data for Atlantic Congo
data = d2[d2$glotto.family=="Atlantic-Congo",]
# use only binary categories
data = data[data$BasicWordOrder %in% c("SVO","SOV"), ]
data = data[data$AdpositionOrder %in% c("Prepositions","Postpositions"), ]


# Make variables for the function:
# Taxon labels
taxlabels = data$iso_code
# Character data (data frame)
characterData = data[,c("BasicWordOrder","AdpositionOrder")]


# Call our function to make the nexus file
makeNexusFile(taxlabels, characterData, "data/Atlantic-Congo.nex")
```

Going even further

The code above means that we could call the `makeNexusFile` function many times, without having to repeat the code within the function. But we can go even further. This function could be quite useful to many other scripts, so we could put the function inside its *own script file*, then load it inside any script where we need it.

For example, we could copy everything in the code above the `DATA` section into another script file (called "makeNexusFileFunctions.R"), then use the `source` function to load it into memory.

Our script to convert data is now much shorter (note that we're also making a file for Uto-Aztecan):

```r
# Load funcitons from another file
source("makeNexusFileFunctions.R")

#############
# DATA

# Load the WALS data
d <- read.csv("data/WALS_WordOrder.csv", stringsAsFactors = F)
glottoData <- read.csv("data/Glottolog_Data.csv", stringsAsFactors = F)
d2 <- merge(d, glottoData, by.x='glottocode', by.y='glotto.code')

###############
# Make a nexus file for Atlantic Congo

# Load data for Atlantic Congo
data = d2[d2$glotto.family=="Atlantic-Congo",]
# use only binary categories
data = data[data$BasicWordOrder %in% c("SVO","SOV"), ]
data = data[data$AdpositionOrder %in% c("Prepositions","Postpositions"), ]


# Make variables for the function:
# Taxon labels
taxlabels = data$iso_code
# Character data (data frame)
characterData = data[,c("BasicWordOrder","AdpositionOrder")]


# Call our function to make the nexus file
makeNexusFile(taxlabels, characterData, "data/Atlantic-Congo.nex")

###############
# Do the same for Uto-Aztecan

# Load data for Uto-Aztecan
data = d2[d2$glotto.family=="Uto-Aztecan",]
# use only binary categories
data = data[data$BasicWordOrder %in% c("SVO","SOV"), ]
data = data[data$AdpositionOrder %in% c("Prepositions","Postpositions"), ]


# Make variables for the function:
# Taxon labels
taxlabels = data$iso_code
# Character data (data frame)
characterData = data[,c("BasicWordOrder","AdpositionOrder")]


# Call our function to make the nexus file
makeNexusFile(taxlabels, characterData, "data/Uto-Aztecan.nex")
```

TASK: Can you see any redundancy in the script above? Let's say you wanted to make a nexus file for *every* language family. Is there a way to put some of this code into a function, then call the function for each language family?