R Reference Material

1. Data Structures

1.1. Data types in R

A. Singles:
- The simplest form of data. Comprised of:
  i. Logicals - Ex., TRUE, FALSE
  ii. Numerics - Ex., 1, 1.2
  iii. Characters - Ex., "5", "abc"

B. Vectors:
- Vectors are a group of singles. However, vectors can only contain one type of single.
- If you force different data types into the same vector, they will be coerced into more basic data types. Ie., logicals -> numerics -> character
- Vectors are the simplest type of data to introduce 'object information', or meta-data. Vectors have a length, and can be named, and store time-series information. The other data types also have these properties.

C. Factor:
- A special type of vector which contains data that can only take on finite values

D. Matrices / Arrays:
- A matrix is a 2-dimensional vector. All elements of the matrix must be of the same data type.
- Unlike vectors, matrices do not have names but do have dimensions.
- Arrays are 3-dimensional matrices. However, because the R console is a 2-dimensional interface, an array is displayed simply as a 'list' of multiple matrices of the same dimensions (but it is not actually a list according to R).

E. Data Frames:
- A data frame is a set of parallel vectors. Vectors need not be of the same data type.
- A data frame containing only numbers will behave similarly to a matrix (in terms of math)
- A data frame is used to cross-reference data; therefore columns should be the same lenth.

F. List:

- The most general data structure in R. Elements can be of different types and lengths and need not be related in any way. Pretty much anything can be represented by a list.
- Most objects returned from built-in functions in R are just complicated lists!

1.2. Operators

1.2.1. Arithmetic operators

`+, -, *, /` - Addition, subtraction, multiplication, division

`%%` - The remainder of vector 1 / vector 2

`%/%` - The quotient of vector 1 / vector 2. If vector 1 is larger than vector 2 it returns 0.

`^` - Exponent

1.2.2. Relational operators

`>, >=` - Checks if each element of vector 1 is 'greater than' / 'greater than or equal to' vector 2

`<, <=` - Checks if each element of vector 1 is 'less than' / 'less than or equal to' vector 2

`==` - Checks if each element of vector 1 is 'equal to' vector 2

`!=` - Checks if each element of vector 1 is 'not equal to' vector 2

1.2.3. Logical operators

`&` - Element-wise 'AND' operator. Returns TRUE only if both elements are true

`&&` - 'AND' operator. Only takes the first element of each vector.

`|` - Element-wise 'OR' operator. Returns TRUE if either element is true

`||` - 'OR' operator. Only takes the first element of each vector.

`!` - 'NOT' operator. Takes each element of a logical vector and assigns i the opposite logical value.

`%in%` - Asks if an element belongs to a vector. It is a shortcut to using `'|'` when repeating variables.

- Ex., `x$name %in% c("A", "B")` is identical to `c(x$name == "A" | x$name == "B")`

1.3. Formulae

Model formulae are *symbolic*. Arithmetic operators take on different meanings in formulae. Therefore, typical arithmetic operations do not apply.

- Note: while arithmetic operators have different meanings, certain arithmetic functions remain the same. For example, the variables can be transformed using the `log()` function.

`~` - Indicates "as a function of."

Ex., `Y ~ X` will create a model where `Y` is a function of `X`.

`+` - Indicates "and." i.e., The addition of another term

Ex., `Y ~ X + Z` means: "`Y` is a function of `X` and `Z`".

`:` - Indicates an interaction term.

Ex., `Y ~ X:Z` means: "`Y` is a function of the interaction between `X` and `Z`".

`*` - Indicates "cross."

Ex., `Y ~ X*Z == Y ~ X + Z + X:Z`

`^` - Indicates "power." This operator expands the interactions between the variables up to a given order (i.e., it includes all main effects plus all pair-wise interactions up to the given order).

Ex., `Y ~ (X + Z + W)^2 == Y ~ X + Z + W + X:Z + X:W + Z:W`

`-` - Indicates "without." Subtracts terms from the formula, if possible (otherwise it is ignored).

Ex., `Y ~ (X + Z + W)^2 - Z:W == Y ~ X + Z + W + X:Z X:W`

`.` - Indicates "every variable found in the data frame in the data argument."

- It may be faster to select a subset of variables and make a new data frame rather than make a loop to impute all of the variables into a formula object.

`as.formula()` - Converts the present object into a formula object (most commonly a string).

`I()` - Overrides the symbolic interpretation of formulae, and invokes the usual arithmetic instead.

Ex., `Y ~ (X * Z)^2 == Y ~ (X + Z)^2`

But `Y ~ I((X * Z)^2) != Y ~ I((X + Z)^2)`

- For constructing polynomials, the `poly(x, n)` function provides an easy way to include polynomial transformations of `x` up to the n^{th} degree.

`paste()` - First converts elements of a vector into characters, and concatenates them into a single element.

`...` - One or more R objects to be converted to character vectors (or is already a character). Multiple terms can be passed for this argument.

`sep` - Character string. What character string should separate the terms in `paste()`? Default = " "

- `paste()` is often useful because you can use it to store arguments to use in a function, allowing for cleaner code (See the exanple in **6.4. Worked Examples**).
- It also allows you take the output of certain functions and then use them as arguments to a function (See the example in **0. Applications of `glm()` - Classification Models and ROC Analysis**.).

2. Manipulating Data

Because the most commonly used object to store and work with data is the data frame, this section focuses primarily on working with data frames.

2.1. Basic knowledge for working with objects

2.1.1. Terminology

'Object' - The piece of data stored in R memory by the user, which can be called upon later. An object is essentially a variable (in the general sense), but it can be complex (i.e., a collection of variables). We reserve the term 'variable' for a specific use when referring to R.

'Variable' - The column of a data frame - essentially a vector. An object made up of a single variable *is* a vector.

'Observation' - The row of a data frame. The term 'observation' is centered around a reference column (ex., patient_ID), but describes all the data in the row associated with the value in the reference column.

'Case' - A subject/participant/experimental condition. A case is distinct from an observation in that one case might have multiple observations associated with it (i.e., for time-series data). For non-panel data (ex., cross-sectional data), the terms 'case' and 'observation' are essentially synonymous - *but try to maintain the distinction*.

'Entry' - The meeting point of a variable and an observation. I.e., a single data point. Sometimes synonymous with the term 'value', but 'entry' is preferred.

2.1.2. Creating matrices, data frames, and tables (basic functions)

`cbind(x, y) / rbind()` - "column bind"; Merges variables into a matrix or data.frame. Whichever variable is listed first will appear first.

`colnames(x) / rownames()` - Returns the column (or row) names of a matrix/data frame. Can be used to define column names colnames(x) <- y

- Note: row/column names on *matrices* are purely descriptive and cannot be used for subsetting like for data frames.

`data.frame()` - Creates a data frame

`...` - Object or data to be used in the matrix

`row.names = NULL` - May described a column in `...` to be used as row names, or you may specify new row names using a character vector.

`stringsAsFactors = default.stringssFactors()` - Still character strings be converted to factors or treated as character strings? The default R setting for this is TRUE.

- This can be changed by setting: `options(stringsAsFactors = FALSE)`

`check.rows` - Check rows for consistency in length and naming

`check.names` - Check column names for syntactically valid variable names and duplicates

`matrix()` - Creates a matrix

`data = NA` - Object or data to be used in the matrix.

`nrow / ncol = 1` - Dimensions of the matrix. Only one needs to be specified in R

`byrow = FALSE` - By default, `data` fills the matrix by column. If TRUE, it will fill by row.

`dimnames = NULL` - Row and column names. A list of length 2 in this argument will give row and column names (in that order)

`table(x, y)` - Creates a contingency table out of variable x and y.

2.1.3. View the properties of objects (basic functions)

`attributes()` - Returns the attributes of a variable

`class()` - Returns the type of variable.

`dim()` - Returns the dimensions of a table

- It can also be used to define dimensions. Ex.,

```
dim(x) <-c(a, b)
```

`head(name, n) / tail()` - Displays the first/last n rows of a table or elements for a vector

`n = 6`

`identical()` - Logical. Tells you if 'x' is identical to 'y'.

`x, y` - The two objects to be tested

- This function is similar to using the operator `x == y`, except it tests each element and returns only one TRUE / FALSE, rather than doing so for each individual element.

`length()` - Returns the number of elements of an object as a vector.

`mode()` - Returns the data mode (i.e., *the single*) of the variable (ex., logical, numeric, character)

`str()` - "structure." Provides a complete summary of the variable including class, dimensions, types of variables, and levels within variables. However, str() is a display function, and its value is NULL. Therefore, you cannot use it to subset data.

`unique()` - Returns the value of unique element of a vector. I.e., returns every value of a vector (once).

2.1.4. Checking and converting between data types

discuss functions for checking a data structure's type as well as functions to coerce data types to other types

2.1.5. The `with()` and `attach()` functions

`attach` - Allows you to temporarily store an object (until you call `detach()`) so that you can make references to specific variables within the object without calling on the object name each time.

- Note: If your object contains a variable that is already in the R memory, then it will be masked by that object. This is because the `attach()` function is stored with less priority than objects stored in the R memory. To get around this, you can use `with()` to temporarily change the priority

`with()` - allows you to temporarily store an object (until the function is closed) so that you can make references to specific variables within the object without calling on the object name each time. For example,

`data` - object that you are storing, typically a data frame or a list

`expr` - the expression that you are evaluating

- Ex.,

```
x <- with(airquality, table(Ozone >80, Month)) #is the same as
x <- table(airquality$Ozone >80, airquality$Month)
```

2.2. Subsetting in R

2.2.1. Worked Examples

This is a quick reference on subsetting in particular situations.

A. Extract data for a variable by another variable.

'In this example we want to extract the data for 'Age', but only for male individuals. Our data frame is named 'x'. In this case, we are "extracting 'Age' by 'Sex'."

Note that this may be more easily solved using the `subset()` function (**Section 2.2.6. subset() function to subset columns**). `subset()` keeps the data in the data.frame class, but is generally not ideal for programming.

```
x <- data.frame(
  Sex    = c("Male", "Female"),
  Age    = rnorm(100, 50, 10),
  Weight = rnorm(100, 140, 30))
# Both of these solutions are possible. The second is more "proper"
x$Age[x$Sex == "Male"]
x[x$Sex == "Male", "Age"]
# We can take build on this. Ex.,
x[x$Sex== "Male" & x$Age > 50 | x$Weight <= 120 , "Weight"]
```

To my knowledge, there is no way to perform extraction by multiple variables while preserving the structure of the data frame (except using the `subset()` function)

2.2.2. Introduction to extracting from data frames

Data frames consist of cross-referenced data, allowing us to subset data more easily. However, depending on what pieces of the data you are trying to extract, subsetting may become complicated.

R's subsetting operators are powerful and fast. Mastery of subsetting allows you to succinctly express complex operations in a way that few other languages can match. Subsetting is hard to learn because you need to master a number of interrelated concepts:

- The three subsetting operators ('[ ]', '[[ ]]', '$')
- The six types of subsetting
- Differences in behaviour for different data structures. especially in simplifying vs preserving structure.
- The use of subsetting in conjunction with assignment.

The following website: http://adv-r.had.co.nz/Subsetting.html provides excellent information on subsetting.

Most techniques used for subsetting data frames can also be applied to other data structures, such as vectors, lists, and matrices. The exception is that data frames have the unique ability that you are able to refer to columns by NAME. The benefit to subsetting by name is that we do not need to look up the column number. Ex.,

This section reviews different functions and techniques that can be employed to subset data in base R and `dplyr`

2.2.3. `'[ ]'`, `'[[ ]]'`, and `'$'` operators for subsetting

Note: When extracting data using '[ ]', the values will be stored in the mode 'list'.

`x[i, j]` - Operator which acts on vectors, matrices/arrays, and lists, to extract or replace parts. Returns a list.

- `x` - Object from which to extract or replace elements
- `i, j, ...` - Indices specifying the elements of object '`x`' to extract or replace. Indices are numeric or character vectors
  - In addition to or rather than the indices '`i`' and '`j`', logical vectors can be used to extract data
- `name` - Character vector which is matched to the names / dimnames attribute).
- `drop = TRUE` - Only applies to matrices/arrays. If TRUE, will coerce the result to the lowest possible dimension. If FALSE, will preserve the same dimensions as the original object. Only applies to extracting elements, not for replacements. When subsetting

`x[[i, j]]` - Operator which acts on vectors, matrices/arrays, and lists, to extract or replace parts. Returns an element.

`exact = TRUE` - Controls partial matching of '`[[ ]]`' when extracting data when '`i`' is a character vector. If TRUE, '`i`' must exactly match the names / dimnames attribute of the object. If FALSE, the function will allow 'partial matching'

- Ex., You can refer to an element named 'Alex' using:
  `x[["A", exact = FALSE]]`

- The '`[[ ]]`' operator is technically the same operator as '`[ ]`'. Therefore, all of the possible arguments are the same. However, the usage is different, and the `exact` argument is only relevant to '`[[ ]]`', so I discuss it here.
- '`[[ ]]`' differs from '`[ ]`' in that it returns a single element of the list rather than the list itself. Therefore, the mode of the result will be a vector.

`x$name` - The '$' is simply shorthand for `[["name", exact = FALSE]]`.

- Note that setting `exact = FALSE` may result in errors from unintended partial matching.
- This is why '$' doesn't use " ".
- The '$' operator is used to subset variables within a data frame based on their names.

2.2.4. Types of subsetting

This assumes you are subsetting with an atomic vector, the simplest data type that you can subset. Different data types allow for more complex variations of this.

1. Logical: Ex., `x[ !is.na(x) ]` returns values of 'x' where is.na is FALSE.
2. Positive integers: Ex., `x[ c(1,3,5), ]` returns the 1st, 3rd, and 5th rows, and all columns of a matrix.
3. Negative integers: Ex., `x[ c(-1,-3,-5), ]` excludes the 1st, 3rd, and 5th rows.
   - This can be used for selection by index and by named elements.
4. Names: Ex., `x[ c("A", "B") ]` returns elements with the name "A" or "B" assigned to them.
5. Nothing: Returns the original vector. Not useful for vectors but is useful for 2D data structures as well as for assignment
6. Zero: Returns a zero-length vector. As far as I can tell it's pretty useless.

2.2.5. Simplifying vs. preserving in different data types

The rules of subsetting are inconsistent across data structures, and it is very difficult to remember how each behaves. This is especially true for *simplifying vs. preserving* the structure of the object when subsetting.

- Preserving structure is generally better for programming because it will always result in the same data type.

- Simplifying structure will return the simplest possible data structure that can represent the output. It is useful for use interactively.

	Default	Simplifying	Preserving
Vector	Preserve	`x[[1]]`	`x[1]`
List	Preserve	`x[[1]]`	`x[1]`
Factor	Preserve	`x[1:4, drop = T]`	`x[1:4]`
Array	Simplify	`x[1, ]` **or** `x[, 1]`	`x[1, , drop = F]` **or** `x[, 1, drop = F]`
Data frame	Depends*	`x[, 1]` **or** `x[[1]]`	`x[1]` **or** `x[, 1, drop = F]`

* See Error! Reference source not found.. Error! Reference source not found..

Preserving is the same for all data types: you get the same type of output as input. Simplifying behaviour varies slightly between different data types. You should consult: http://adv-r.had.co.nz/Subsetting.html#simplify-preserve for reference for this.

1. Atomic Vectors:
   - '[ ]' can be used in the ways discussed in **2.2.4. Types of subsetting** to extract elements from a vector

2. Lists:
   - More or less works the same was as with a vector.
   - '[ ]' will always return a list.
   - '[[ ]]' will return an element inside of a list

3. Matrices/arrays:
   - Higher dimensional structures can be subset in three different ways:
   - With a single vector
     - Ex., `x[1, 2]` will return the element in the first row and second column
   - With multiple vectors
     - Ex., `x[c(1,4), c(2,3)]` returns every combination of row × column. If 'x' is a matrix, it will return [1,2], [1,3], [4,2], [4,3].
   - With a matrix
     - Each row and column of this matrix corresponds to the row and column index locations of the matrix being subsetted. A 2-column matrix should be used to subset a matrix, while a 3-column matrix should be used for an array.
     - Ex., `x[ matrix(ncol = 2, byrow = TRUE, c( 1, 1, 3, 1 )]` will return [1,1], [3,1]
     - Ex., `x[cbind(c1,4), c(2,3)]` returns [1,2], [4,3]

4. Data frames:
   - Data frames possess the characteristics of both lists and matrices. If you subset with a single vector it will behave like a list; if you subset with two vectors, it behaves like a matrix. This is important because matrices simplify by default and lists do not.
   - `x[i]` behaves like a list
   - `x[i, j]` behaves like a matrix

5. S3 objects: See http://adv-r.had.co.nz/Subsetting.html#data-types

6. S4 objects: See http://adv-r.had.co.nz/Subsetting.html#data-types

2.2.6. `subset()` function to subset columns

`subset()` - This function can be used to retrieve a subset of your data containing columns of interest and rows of interest within those columns. However, `subset()` loses the data's context, so it cannot be placed back into the original data frame.

`x` - The object of interest

`select` - An expression used to select the columns of interest.

`...` - The argument used to subset the data

- This argument is logical that is used to subset the columns further (i.e., to select rows of interest) by its value.
- Note your subsetting must be done in a single logical argument, so conditional operators should be used if there are multiple conditions. Ex.,

```
subset(iris, Species == "setosa" & Sepal.Length <
median(iris$Sepal.Length), select = -(Species) )
```

- This function is similar to `filter()` in `dplyr`
- Note: When using `filter()`, in `dplyr` observations will retain the same row numbers as the original data frame. When using `subset()`, the observations will be re-numbered (starting from 0). All of the values are the exact same, but the row number should not be used to cross reference these data.
- Caution: *"This is a convenience function intended for use interactively. For programming it is better to use the standard subsetting functions like [, and in particular the non-standard evaluation of argument `subset` can have unanticipated consequences."* - R help page

2.3. Other ways of manipulating data

2.3.1. Deleting data from a data frame

Data can be deleting by subsetting the desired data and setting it to `NULL` (or alternatively some other value) Ex.,

```
dat$useless_column <- NULL
```

2.3.2. Working with NA values

1. Filter out NAs by displaying only values that are not NA. Ex.,

```
airquality[!is.na(airquality$Ozone), ] #OR
subset(airquality, !is.na(Ozone)) #OR
airquality[na.omit(airquality$Ozone), ]
 # Note that this method will filter out any observations which contain NA for the column for which the argument was passed (i.e., Ozone). This will remove observations from other variables even if they were not NA values.
airquality[!complete.cases(airquality), ] # OR
na.omit(airquality)
 # Returns only observations for which all data are available
```

2. Replace NA values with another value. Ex.,

```
data <- dplyr::mutate(data, Col1 = ifelse(is.na(Col1), 0, Col1))
 # For some reason this above code worked when is.na() did not.

dat[is.na(dat)] <- 0
 # The opposite can also be done:
dat[dat == 0] <- NA
```

3. Summary functions (ex., sum(), mean()) have a built-in argument `'na.rm = TRUE'` that will ignore NA values.

2.3.3. Merging two data frames

Use when you have two data frames describing the same item (ex., two sets of data describing a single set of patients). While you may use the simple `cbind()` command, the observations may not be cross-referenced properly if the observations are not in the same order.

The `merge()` function looks for similar columns, and then merge two data frames with reference to those columns. Ex., for patient data, `merge()` will find that `patient_ID` is the same across both data frames (even if it is not in the same order) and merge the data frames so that all the values remain matched to the proper `patient_ID`.

`merge()`

`x, y` - The two data frames to be merged

`by = intersect(names(x))` - Which column (i.e., column name) should be used as a reference to merge the data? R will automatically look for reference columns, but it is less error-prone if you enter this yourself.

`by.x, by.y = by` - If the reference column is named differently between the two data frames, the 'by' argument can be applied to the data frames independently of each other.

`all = FALSE` - Logical. Should all the data be included (i.e., TRUE), or only data that are able to match (i.e., FALSE)?

`all.x, all.y = all` - Logical. 'all' argument can be applied to either data frame independently.

- Note that R sorts the data according to the first column used in `by` and organizes the rows so that the first data frame (i.e., `x`) is on top.

2.3.4. Ordering data in a data frame

`sort()` - This is the same thing as `order()` except it only works with a single variable. When a single argument is passed, `sort()` will return the value of the elements. In contrast, `order()` will return the index position.

`order()` - Rearranges data frames into ascending or descending order based on the input variables. In the event of a tie, the second variable (etc.) will be used to break the tie. In the absence of a second variable, the index position will be used.

`...` - Vector (most commonly a variable in a data frame). The object to be sorted.

`x` - Atomic vector (only used if sorting a single vector).

`partial` - Vector of indices for partial sorting

`decreasing` - Logical. Should the sort order be decreasing? Default = FALSE

`na.last` - Should NAs be sorted to be put last? If FALSE, they are put first. If 'NA', they are removed.

- *Note*: <u>The variables in `...` need not be from the same data frame</u>! `order()` will sort observations by their index positions.

`dplyr::rearrange()` - Works in a similar manner to **order()**. **See 11.1. Data manipulation in `dplyr` (basic functions)**

<u>Brief Examples</u>

```
y <- data.frame( Variable1 = c( 6, 6, 7, 5, 7), Variable2 = c( 9, 6, 7, 8, 5) ); y
  Variable1 Variable2
1         6         9
2         6         6
3         7         7
4         5         8
5         7         5
```

```
x <- c(9, 7, 0, -2, 1)
```

```
sort(x)    # Note that it outputs the element values
[1] -2  0  1  7  9
order(x)  # Note  that it outputs the element index
[1] 4 3 5 2 1
```

```
y[ order(y$Variable1), ]    # Note that ties in Variable1 are broken by index position.
  Variable1 Variable2
4         5         8
1         6         9 <- Tied 1:2
2         6         6 <- Tied 1:2
3         7         7 <-- Tied 3:5
5         7         5 <-- Tied 3:5
```

```
y[ order(y$Variable1, x), ]    # Note that x is not part of data frame y; y is sorted based on the value of order(x)
  Variable1 Variable2
4         5         8
2         6         6 <- 3 in order(x)
1         6         9 <- 4 in order(x)
3         7         7 <-- 2 in order(x)
5         7         5 <-- 5 in order(x)
```

2.4. Tables: Grouping data by a grouping variable

2.4.1. `aggregate()`, `by()`, and `tapply()`

Use when you have data that can be sorted using a grouping variable (ex., patient, control). There are several functions that can be used to do this, including `aggregate()`, `by()`, and `tapply()`.

Note: For large data sets (>100,000 elements) `tapply()` will have the shortest run-times.

`aggregate()` - Groups observations by a grouping variable, then output a data frame with the desired statistic for the different groups.

- `x` - An R object. Either a vector or data frame.
- `by` - What is the grouping variable? This must be of mode `list`. Either co-erce the variable using as.data.frame() or use named index selection with the '[ ]' operator (see example)
- `FUN` - What function should be used to compute the summary statistic of interest (Ex., `mean`)

`simplify = TRUE` - If possible, should the output be simplified to a vector or matrix?

`formula` - The formula to decide how the variables wilbe grouped. Generally it will follow the form '`y~x`', which means, "group column `y` by column `x`'

`data` - The data frame (or list) from which the variables should be taken

`subset` - An optional vector specifying which subset of observations should be used

`na.action` - How should NA values be handled? By default, they are ignored.

- Ex., **Usage 1: 'data.frame' method**

```
aggregate( ASC$ASC_Speck, by = ASC["Treatment"], FUN = sum) # OR
aggregate( ASC$ASC_Speck, by = as.data.frame(ASC$Treatment), FUN = sum)
```

```
  Treatment   x
1         1  45
2         2  35
3         3 108
```

- Ex., **Usage 2: 'formula' method**

```
aggregate(ASC_Speck~Treatment, data = ASC, FUN = sum)
```

```
  Treatment   x
1         1  45
2         2  35
3         3 108
```

`by()` - `by()` is a wrapper for `tapply()` applied to data frames. It will output a numeric with the class 'by'.

`data` - An R object. Usually a data frame, possibly a matrix

`INDICES` - A factor or list of factors each with length equal to the number of rows in `data`. In other words, `length(index) == nrow(data)`

`FUN` - What function should be used to compute the summary statistic of interest (Ex., `mean`)

`simplify = TRUE` - Logical. If TRUE, then `FUN` will return a scalar. If FALSE, `tapply()` will output an array with the mode 'list' (i.e., a list with a '`dim`' attribute)

- Ex.,

```
by(ASC$ASC_Speck, INDICES = ASC$Treatment, FUN = sum)
```

```
ASC$Treatment: 1
[1] 45
------------------------------------------------------------
ASC$Treatment: 2
[1] 35
------------------------------------------------------------
ASC$Treatment: 3
[1] 108
```

`tapply()` - "tabulate apply". Groups observations by a grouping variable, then output a vector with the desired statistic for different groups.

`x` - An atomic object. Usually a vector.

`INDEX` - A list of one or more factors, each with the same length as '`x`'

`FUN` - What function should be used to compute the summary statistic of interest (Ex., `mean`)

`simplify = TRUE` - Logical. If TRUE, then `FUN` will return a scalar. If FALSE, `tapply()` will output an array with the mode 'list' (i.e., a list with a '`dim`' attribute)

- Note: Arguments for the function `FUN` can be included as arguments for `tapply()`. Ex., `mean()` by default does not ignore NA values, but has the `na.rm` argument which can be passed as TRUE in `tapply()` to remove them.
- Ex., `tapply()` usage

```
tapply(ASC$ASC_Speck, INDEX = ASC$Treatment, FUN = sum)
  1   2   3
 45 35 108
```

`dplyr::group_by` combined with `dplyr::summarise()` can similarly be used to aggregate and summarise data, with faster run times than `tapply()` for large data sets. `summarise()` also makes it easier to compute more than one summary statistic versus base R. **See 11.1. Data manipulation in `dplyr` (basic functions).**

2.4.1.1. `aggregate()`, `by()`, and `tapply()` for multiple grouping variables

Use when you have data that need to sorted using multiple grouping variables simultaneously (ex., patient vs. control, and male vs. female). The same functions that are used in **0. In the event of a tie,** the second variable (etc.) will be used to break the tie. In the absence of a second variable, the index position will be used.

`...` - Vector (most commonly a variable in a data frame). The object to be sorted.

`x` - Atomic vector (only used if sorting a single vector).

`partial` - Vector of indices for partial sorting

`decreasing` - Logical. Should the sort order be decreasing? Default = FALSE

`na.last` - Should NAs be sorted to be put last? If FALSE, they are put first. If 'NA', they are removed.

- *Note*: The variables in `...` need not be from the same data frame! `order()` will sort observations by their index positions.

`dplyr::rearrange()` - Works in a similar manner to **order()**. **See 11.1. Data manipulation in `dplyr` (basic functions)**

Brief Examples

```
y <- data.frame( Variable1 = c( 6, 6, 7, 5, 7), Variable2 = c( 9, 6, 7, 8, 5) ); y
  Variable1 Variable2
1         6         9
2         6         6
3         7         7
```

```
4        5        8
5        7        5
```

```
x <- c(9, 7, 0, -2, 1)
```

```
sort(x)   # Note that it outputs the element values
[1] -2  0  1  7  9
order(x)  # Note  that it outputs the element index
[1] 4 3 5 2 1
```

```
y[ order(y$Variable1), ]   # Note that ties in Variable1 are broken by index position.
  Variable1 Variable2
4         5         8
1         6         9 <- Tied 1:2
2         6         6 <- Tied 1:2
3         7         7 <-- Tied 3:5
5         7         5 <-- Tied 3:5
```

```
y[ order(y$Variable1, x), ]   # Note that x is not part of data frame y; y is sorted based on the value of order(x)
  Variable1 Variable2
4         5         8
2         6         6 <- 3 in order(x)
1         6         9 <- 4 in order(x)
3         7         7 <-- 2 in order(x)
5         7         5 <-- 5 in order(x)
```

Tables: Grouping data by a grouping variabl can be used when there are multiple grouping variables with slight modifications to the arguments.

- in `aggregate()` simply adjust the `formula` argument with a '+'
- in `by()` and `tapply()` the indices must be given as a list. Use `list()`.
- in `dplyr::group_by()` simply pass another variable.

Examples (data are not real):

```
aggregate(ISO~Sex+Group, data = dat, mean)
     Sex Group      ISO
1 Female    BD 52.28197
2   Male    BD 50.63031
3 Female   CTL 51.14350
4   Male   CTL 50.77194
# Output is a data frame
```

```
tapply(dat$ISO, INDEX = list(dat$Group, dat$Sex), mean)
      Female     Male
BD  52.28197 50.63031
CTL 51.14350 50.77194
```

Output is a matrix

2.4.2. `cut()` to group data by a continuous variable

Continuous variables can be broken up into quantiles using the `cut()` function (i.e., converting it to ordinal data). The data can then be subjected to `aggregate()`, `by()`, `tapply()`, or `table()`.

`cut()`

`x` - Numeric vector to be cut.

`breaks` - Breaks can be assigned in two ways. 1) Enter a single integer ≥ 2 corresponding to the number of bins (Ex., for quartiles enter '4'). 2) Enter a numeric vector with two or more cut points, corresponding to the element where the cuts should be made.

`labels = TRUE` - Labels for the levels of the resulting category. By default, labels display the range of the interval. If FALSE, simple integer codes (corresponding to the quantile) are returned instead.

`right = FALSE` - Logical. Should intervals be right-open/left-closed? Ex., (0,1] denotes $0 < x \leq 1$

- Ex.,

```
my.cut <- cut(airquality$Wind, breaks = 5, label = FALSE)
tapply(airquality$Solar.R, my.cut, mean, na.rm=TRUE)
      1       2       3       4       5
195.3333 190.5472 185.3654 180.1154 127.0000
```

2.4.3. `table()`, `prop.table()`, and other basic functions

Tables are another way of grouping data by a categorical variable. Tables are generally used to express counts or frequencies.

- `table()` - Table of counts (generally the data are not grouped)
- `xtabs()` - Table of counts using a formula (generally the data are not already grouped)
- `prop.table()` - Frequency table
- `addmargins()` - Use to add calculate summary statistics for the rows / columns

`addmargins()` - Puts margins on a a table. For example, it can be used to display the sums or means of each column/row

`A` - The table (or array) to which the margins will be appended. The function uses the presence of the '`dim`' and '`dimnames`' attributes of '`A`'.

`margin = seq_along(dim(A))` - A vector containing the dimensions over which to form the margins. Margins are formed in the order in which dimensions are specified in this argument. By default, it takes on the dimensions of the table in '`A`'

`FUN = sum` - A list with the same length as `margin`, with each element of the list either being a function (or a list of functions). The names of the elements (i.e., functions) in this vector will appear as levels in the '`dimnames`' attribute of the output.

`quiet = FALSE` - Logical. By default, the function will print a message that tells you the order in which margins were computed.

`ftable()` - Creates a 'flat' contingency table. Useful when more than two variables are being cross-tabulated at once.

`x` - Object that will be used to construct the table. The object must be able to be interpreted as a factor.

- Ex.,

```
with(airquality, ftable("temp>65" = Temp > 65, Month, "Day>16" = Day > 16))
```

`margin.table()` - Computes the sum of the counts/frequencies contained in the table. It will add them up along the rows, columns, or other (if you have a higher-order table)

`x` - an array

`margin` - See `margin` argument in `prop.table()`

`prop.table()` - Converts a table into a frequency table (i.e., expressing tables as fractions of a marginal table)

`x` - The table to be converted

`margin = NULL` - Vector or vector of indices. Which margin should the probabilities be calculated relative to?

- `NULL`: Calculate the joint probability (Ex., the absolute probability of $x_1 \times y_1$ occuring)
- `1`: Calculate probability relative to the marginal probability of 'variable 1' (i.e., rows add up to 1.000)
- `2`: Calculate probability relative to the marginal probability of 'variable 2' (i.e., columns add up to 1.000)
- Higher order margins can be used as inputs if the table is composed of more than 2 cross-referencing variables. (Ex., 3 refers to 'variable 3')

`table()` - Using one or more cross-referenced factors, it builds a contingency table of the counts of class "table".

`...` - One or more objects that will be used to construct the table. The objects must be able to be interpreted as factors (including character strings); this usually comes from a data frame but may also include lists.

- If more than one object is passed, the data should be cross-referenced in some way, or else the data will be matched by index position (which is meaningless)

`exclude` - levels o remove for all factors in '`...`' (I don't know what this means)

`useNA = "ifany"` - Should the table count NA values? Takes on values: "no", "ifany", and "always"

- Both `exclude` and `useNA` are automatically apply to all of the objects in '`...`' There are ways that allow you to modify how the table interacts with NA values for each object separately, but they are more work and I don't know them. Look it up if it's ever a problem.

- Ex.,

```
table(airquality$Temp > 65, airquality$Month)
```

`xtabs` - Creates a contingency table based on a formula and data (rather than counts).

`formula = ~.` - A formula for how to group the variable.

`data` - The matrix or data grame containing the variables in `formula`

- This function works in a similar way to the formula method of `aggregate()`. See **0. In the event of a tie,** the second variable (etc.) will be used to break the tie. In the absence of a second variable, the index position will be used.

`...` - Vector (most commonly a variable in a data frame). The object to be sorted.

`x` - Atomic vector (only used if sorting a single vector).

`partial` - Vector of indices for partial sorting

`decreasing` - Logical. Should the sort order be decreasing? Default = FALSE

`na.last` - Should NAs be sorted to be put last? If FALSE, they are put first. If 'NA', they are removed.

- *Note*: <u>The variables in `...` need not be from the same data frame</u>! `order()` will sort observations by their index positions.

`dplyr::rearrange()` - Works in a similar manner to **order()**. **See 11.1. Data manipulation in `dplyr` (basic functions)**

<u>Brief Examples</u>

```
y <- data.frame( Variable1 = c( 6, 6, 7, 5, 7), Variable2 = c( 9, 6, 7, 8, 5) ); y
  Variable1 Variable2
1         6         9
2         6         6
3         7         7
4         5         8
5         7         5
```

```
x <- c(9, 7, 0, -2, 1)
```

```
sort(x)   # Note that it outputs the element values
[1] -2  0  1  7  9
order(x) # Note  that it outputs the element index
[1] 4 3 5 2 1
```

```
y[ order(y$Variable1), ]   # Note that ties in Variable1 are broken by index position.
  Variable1 Variable2
4         5         8
1         6         9 <- Tied 1:2
2         6         6 <- Tied 1:2
3         7         7 <-- Tied 3:5
5         7         5 <-- Tied 3:5
```

```
y[ order(y$Variable1, x), ]   # Note that x is not part of data frame y; y is sorted based on the value of order(x)
  Variable1 Variable2
```

```
4         5         8
2         6         6 <- 3 in order(x)
1         6         9 <- 4 in order(x)
3         7         7 <-- 2 in order(x)
5         7         5 <-- 5 in order(x)
```

- Tables: Grouping data by a grouping variabl.
- Ex.,

```
xtabs(Freq ~ Dept + Admit, data = as.data.frame(UCBAdmissions))
```

2.4.3.1. Subsetting tables

Subsetting tables works differently from other objects. We still use the '[ ]' operator, but it works differently.

- The position of the elements refers to the variable number (the order in which variables were assigned when the table was made) rather than the dimension (row vs. column).
- The value of the element refers to which margin it is referring to (ex., 1 = rows, 2 = columns, 3 = variable 3)

2.5. Manipulating Factors

2.5.1. Defining new factor levels

Below are three strategies (*#2 is probably the proper way)*:

1) Re-code the factor to a string, add the level, and re-code to a factor:

```
x <- as.character(x)
x[is.na(x)] <- "None"  ## Enter whatever subset you want here
x <- as.factor(x)
```

2) Add a factor level directly:

```
x <- factor(x, c( levels(x), "None")
x[is.na(x)] <- "None"
```

3) Convert to integer (note this loses the level names):

```
x <- as.numeric(x)
x[is.na(x)] <- 0
x <- as.factor(x)
```

3. Functions, Loops, and Flow Controls

3.1. function()

`function( arglist ) return(value)`

`arglist` - All of the variables that can be fed into the function. All of the variables must be used. You can define default values inside function().

`return` - Tells the function to evaluate the expression '`value`'. If `return` is not explicitly called, R will return the evaluation for the last expression passed. *I don't know what this really means, but in practice you need to explicitly call* `return` *in some cases. See* **3.1.1.1. Re-coding NAs for factors**.

`value` - The expression that is evaluated given your variables

3.1.1. Worked Examples

3.1.1.1. Re-coding NAs for factors

A function for re-coding factors with NA values to some other value. This function takes data frame `data` and the index position for a variable (`x`) as its inputs. It defines a new factor level "`None`" to all variables and assigns all NA values to this level.

```
impute.na <- function(data, x) {
  levels(data[, x]) <- c(levels(data[, x]), "None")
  data[, x][is.na(data[, x])] <- "None"
  return(data[, x])
}
```

A simple way to apply this is in a `for()` loop. This applies this function to every variable in a data frame `test`.

```
for (i in seq_along(test)) {
  test2[, i] <- impute.na(test, i)
}
  ## Note: if you don't explicitly call return, every observation in test2 will be "None"
```

3.2. Loops

`for(i in 1:10) expression`

- This for() loop will repeat the given expression for each value of *i* from 1 through 10

- for() loops should be somewhat straightforward, but there is always some small error that causes it to fail
-

`while()`

`repeat()`

3.3. `apply()` functions

Fun fact: arguments for the function within the apply() can be passed as arguments in apply(). Example:
`sapply(x, FUN = mean, na.rm = TRUE)`

4. Reading in Data

4.1. Files and directories (basic functions)

`getwd()` - "get working directory." Tells you what directory your current R session is using.
`setwd()` - "set working directory."
`dir()` - Lists the files within the specified directory (default = working directory)
`ls()` - Lists the objects currently used in this R session
`rm()` - Removes the specified objects. Can be used on list=ls() to remove all objects.
`dir.create()` - Creates a new directory (folder) within the working directory
`unlink()` - Deletes a directory
`file.create()` - Creates a new file within the working directory
`file.rename()` - Renames the file
`file1` - Original name of file
`file2` - Desired name for file
`file.copy()` - Similar arguments to file.rename(). Makes a copy of file1 under the name of file2.
`file.path()` - Contructs the path to a file
`file.info()` - Grabs information about the file
`args()` - tells you the arguments used in a function
`name` - Name of the function without '()'

4.2. Reading in data (basic functions)

* Note: RStudio also has a built-in tool for reading in data. Tools > Import Dataset

`read.table()` - Reads a .txt file into a data.frame. Key arguments:

`file` - Files can be read into R in the following ways:

- From your computer using a direct filepath (using '/' [not '\'] between folders)
- From your computer, within your working directory, simply by supplying the file name.
- Online, with a URL using HTTP or HTTPS protocols.

`sep = ""` - What character denotes separation of data points. Default is 'space/tab'.

`dec = "."` - What character is used as a decimal separator?

`header = FALSE` - Is the first line a header (ie., column name)?

`nrows = -1` - How many rows of data are to be read? If this takes on a non-default value, it will cut any rows in excess of the number defined at the end of the data.frame.

`skip = 0` - How many rows are to be skipped *at the beginning* of he table before reading?

`na.strings = "NA"` - What character string denotes a missing value in the original file?

`comment.char = "#"` - Is there a character at the beginning of a line that indicates that the line should be skipped (ie., what denotes a comment)?

- Note: comment strings don't count toward the line count in 'nrows'

`read.csv()` - Reads a .csv file; it uses the same arguments as read.table

`header = TRUE`

`sep = ","`

- Note when saving a .xls file to .csv, only the active sheet will be saved

`read.fwf()` - Reads a table of fixed width format. I don't understand how it works.

`scan()` - Reads data into a vector (or list), instead of a data.frame. Unlike readLines(), each line can have multiple vectors if a separator is defined.

`what = double()` - I don't really understand this argument, but it seems to help if what = " "

`nlines = 0` - Essentially the same as 'nrows' in read.table()

`readLines()` - Reads a .txt file into a vector of characters, where each line of the text is converted into a string.

- If you have numbers on one line and text on another, you can convert your numbers back into a vector using `as.numeric( strsplit( x[1], " "))[[1]]`.
  A. `x[1]` simply indicates the lines which contain our numbers
  B. `" "` indicates that they are separated by spaces.

C. `[[1]]` indicates that we are constructing a vector rather than a data.frame (`[1]`)?

`file()` - Function used to create, open, and close connections. Allows you to read in different sections of a file at a time (allowing you to open them in different ways).

`open = " "` – Indicates how the file should be opened. "`r`" indicates 'open for reading in text mode.' To see the other modes, see ?file.

- Ex., You can open a connection and then read the first 3 lines using scan(). If you proceed to use readLines(), it will automatically take over from the 4th line.
- The connection must be closed using the close() function

4.3. Worked examples

4.3.1. DAT209x Assignment 5:

"The file is a text file and you can open it using a text editor such as Notepad to view its data. You want to read the data from the text file and create a data frame from the data on the text file. The problems are two-folds:

- First, the data are surrounded by lines of text
- Second, they are in different formats.

A good way to handle it is to open a file connection, and read in the data sequentially. For each of the three parts of data, we must specify what deviates from the standards for read.table(), which you can check with ?read.table."

This file contains a lot of data in different formats.

The first set of data is this:

0.45 32 car 0.003
NA 16 bus TRUE
3.14 85 train VALUE
% and then four more lines:
84.34823 666 hoovercraft cosine,sine
40932 666 space_shuttle integral
488.33 666 submarine matrix
2.718 007 bicycle 4.136e-15

But there are more data here:

3,333;45;feet;1
;16;aeroplane;subspace

and then,

some more data:

1.00,-9999,Scooter,4pi
3.21,87,motorcycle,cone

Thats it!

Interpretation:

- 4 lines of unwanted text
- 7 lines of data
  - header = FALSE
  - na.string = NA
  - comment.char = %
  - sep = " "
- 3 line of unwanted text
- 2 lines of data
  - sep = ";"
  - na.string = " "
  - dec = ","
- 5 lines of unwanted text
- 2 lines of data
  - sep = ","
- 2 line of unwanted text

Solution:

```
f1 <- file("Assignment5.txt", open = "r")
x1 <- read.table(f1, skip = 4, comment.char = "%", nrows = 7, na.string = NA)
x2 <- read.table(f1, skip = 3, sep = ";", dec = ",", nrows = 2, na.string = "")
x3 <- read.table(f1, skip = 5, sep = ",", nrows = 2, na.string = "-9999")
# "-9999" is commonly used to denote missing values.
close(f1)
x <- rbind(x1, x2, x3)
# Double check that the data.frame prints correctly and that each column is classified correctly (ie., not as a "character")
x
tapply(x, class)
```

5. Writing Data to Text

5.1. General Usage

Data from R can be exported to many different softwares/file formats:

- Excel, .txt
- SAS, SPSS, STATA - require the 'foreign' package

No matter what format you export to, you generally need to take care of the same things; i.e., it is not easier to export to on format versus another. For this reason, you can simply get away with exporting to .txt files, since all of the other softwares know how to import .txt.

- To write to a .txt, generally use `write.table()`.
  - `cat()` and `writeLines()` are useful if you need to annotate your document (although it may be easier to manually do this on the .txt)
- To save for use on R, use `save()` and `load()`.
- `dump()` redirects your output to a .txt, allowing you to examine the output more closely or send it off.
- Pay close attention to the default arguments of each function.
- Note that R has a precision of 15 decimal places.
  - Ex., `1.0 - 0.9 - 0.1` gives the result `-2.7755575615628914e-17`
  - This will result in minute differences. If greater than 15 digits are required, I think R can handle it, but I don't know how. You will need to look into it.

5.2. Writing data (basic functions)

`write.table()`

`x` - The name of the data.frame/matrix to be exported

`file` - Files should be written into your computer:

- Using a direct filepath (using / between folders)
- By supplying the file name, to save the file in the working directory

`row.names = TRUE` - Are the rows named? When making data in R, you usually want this as FALSE

`col.names = TRUE` - Are the columns named? Note that this is TRUE by default, whereas it is FALSE by default in read.table().

`sep = " "`

`append = FALSE` - If TRUE, will append data.frame 'x' at the end of the specified .txt file (which must already exist). If FALSE, R will overwrite the existing file with the data.frame 'x' or create a new one if it does not exist already.

- An alternative approach is to read in the .txt file using read.table() and then using the rbind() command to append the two data.frames. The file can then be re-written in.

`write.csv()` - Writes a .csv file; it uses the same arguments as write.table

```
sep = ","
dec = "."
```

`cat()` - We previously used this function to concatenate and print a vector. Here we can use it to write to files by specifying the output as a file. The cat() function is ideal for exporting numbers along with an explanation. However, you can also do this by manually entering the comments in the .txt file

- Ex., `cat("Test file for cat\n", rnorm(5), "\n")` will save the two lines of text as a table.

`writeLines()` - Works as the reverse of the readLines() function. writeLines() takes a character vector as an argument and writes each vector as a line in the file.

`sink()` - Used to divert the output of R commands to a new file, rather than the R command prompt. This is useful for debugging large scripts, or working on calculations without bothering to save them as objects. sink() re-directs the R output to a file connection.

- Ex., In this example, everything between the first sink() and second sink() is redirected to the new file and not printed in the command prompt.

```
sink("sinktest.txt")
x <- 1:5
y <- 1:3
outer(x,y)
sink()
```

`dump()` - Used to save R objects to a separate file rather than the R output (as sink() does). The objects look-more-or-less the same in their new file, and can can be sourced and read in R again using the `source()` command.

- Note: Numbers found in the dumped file will have an 'L' attached to them (Ex., 2L). The L indicates to R that the number is an integer, rather than a character vector.
- Note: If objects are generated using RNG and saved using dump(), only the numbers will be saved. When sourcing the object back into R, the object will take on the exact same values.
- Note: To save text representations of objects with dump(), it is required that you actually write the code that generates them. A simpler way of storing text is with the `dput()` function.

`dput()` - Used to save R objects as text. Creates and ASCII representation of any R object. Not 100% sure how/when to use this.

- Ex.,

```
my.list <- list(x = 1:5, y = 3, z = c("a", "b", "c"))
dput(my.list, file = "dputtest.txt")
```

Which will save a file containing:

```
structure(list(x = 1:5, y = 3, z = c("a", "b", "c")), .Names = c("x",
"y", "z"))
```

Note how the file contains no reference to the original my.list object. Instead, it just saves its contents.

`dget()` - Inverse of dput(). However, because dput does not save any reference to the original R object 'x', dget() does not restore 'x'; instead, it simply restores the value of 'x'

`file()` - Function used to create, open, and close connections. Allows you to write in different sections of a file at a time.

`open = " "` - Indicates how the file should be opened. "`w`" indicates 'open for writing in text mode.' To see the other modes, see ?file.

- Ex., Note that file() is assigned to an object f2 and then closed using close(f2)

```
f2 <- file("testfile.txt", open = "w")
mat <- matrix(rnorm(12), ncol = 3)
cat("Header of file \n \n", file = f2)
write.table(mat, file = f2, row.names = FALSE, col.banes = FALSE)
close(f2)
```

`save()` and `load()` - Used to save an R workspace (ie., all of the objects). This is the way to go if you want to transfer a project from one computer to another, but it is only compatible with R.

- Ex.,

```
save(x, y, z, file = "test.RData")
load(file = "test.RData")
```

- This is built-into the GUI of RStudio

5.3. Worked Examples

5.3.1. DAT209x Exercise 6.1

"Create two data frames with the following commands:

```
set.seed(9007)
my.data<-data.frame(x=rnorm(10),y=rnorm(10)+5,z=rchisq(10,1))
additional.data<-data.frame(x=rnorm(3),y=rnorm(3)+5,z=rchisq(3,1))
```

1. Write `my.data` to a file named `Exercise 6.txt`, without row and column names.
2. You suddenly realize that you have more data that you need to write to the file. Consult `?read.table`, and figure out a way to add the contents of the `additional.data` to the contents of the file `Exercise 6.txt`."

Solution:

1.

```
write.table(my.data,"Assignment6.txt", row.names = F)
```

2. A study of `?read.table` reveals that the option `append=TRUE` allows data to be appended to an existing file. Thus, we can issue the command:

```
write.table(additional.data,"Data/Exercise6.1.txt",row.names=FALSE,col.names=FALSE,append=T)
```

Here is an alternate solution:

```
x1 <- read.table("Assignment6a.txt", header = T)
x2 <- rbind(x1, additional.data)
write.table(x2, "Assignment6a.txt", row.names = F)
```

5.3.2. DAT209x Exercise 6.2

"Create a data frame with the following command:

```
set.seed(45)
my.data<-data.frame(x=rnorm(10),y=rnorm(10),z=rnorm(10))
```

1. Write the `my.data` to a csv file using `write.csv2()`. If your regional settings determines that `write.csv()` is more compatible with Excel standards rather than `write.csv2()`, use `write.csv()` instead.
2. Open the file with Excel, save it as an xlsx workbook, and confirm that the file export has happened as it should."

Solution:

```
write.csv(my.data, "testcsv.csv")
```

5.3.3. DAT209x Exercise 6.3

"1. Write the following data to a file named `Exercise 6.3a.txt`:

```
"a";"b"
"A";1
"B";2
"C";3
"D";4
"E";5
```

2. Write the following data to a file named `Exercise 6.3b.txt`:

```
TITLE extra line
2 3 5 7
11 13 17
One more line"
```

Solution:

1.

```
x3 <- data.frame(a = c(LETTERS[1:5]), b = c(1:5) )
```

Note that the first column will be saved as a "factor" rather than a "character". This can be fixed by defining: `x3[ , 1] <- as.character(LETTERS[1:5]` but when the file is written and read back to R, it will be reverted back to a "factor".

```
write.table(x3, "Exercise 6.3a.txt", sep = ";", row.names = F)
```

2.

```
cat("TITLE extra line \n", c(2, 3, 5,7), "\n", c(11, 13, 17), "\n One more line",
file = "Exercise 6.3b.txt")
    # OR
my.text <- "TITLE extra line\n2 3 5 7\n11 13 17 \nOne more line"
writeLines(my.text, con = "Exercise 6.3b.txt")
```

6. Structured Query Language (SQL) databases

SQL is a programming language designed for querying data in "relational databases." Relational databases are a widely used way of storing data, and have largely replaced other structures such as "hierachical" or "network" databases.
Their strength lies in handling a huge volume of data and complicated database structures, where relations between contents cannot be specified in the standard data.frame manner. To query data in such databases, you need SQL.

6.1. General usage

SQL databases can be *very large*, and loading such a database will trigger bandwidth and memory problems. You cannot load such databases into R. For such databases, you want to load in only the relevant data without downloading the entire database.
The `RODBC` package is used to interact with SQL databases (and other formats) into R.
An SQL database may be thought of as a large database of tables. Tables can be thought of as data.frames (but this is not always true).

- First, open the connection to the SQL database using `odbcDriverConnect()` and store it as an object.

To do this you will need information about the server, database, username, password, etc.

A. Use `sqlTables()` and store it as an object. This command will give you a data.frame of ALL the tables contained inside the SQL database alon with some basic information about them including the table schema.
   - As this may be large, it is recommended to view it using head().
   - Use `sqlFetch()` to fetch a single table of interest. Note that you will need to determine the schema for the table from the output of `sqlTables()`.

B. Use `sqlQuery()` for more complex querying and to open large tables. See the sqlQuery() function and the worked examples.

6.2. Basic functions for interacting with SQL databases

`odbcDriverConnect()` - Open a connection to an Open Database Connectivity (ODBC) database (ex., SQL).

`Server` - Name of the server. If stored on your local machine, the default value is the name of your machine. Run-time may be improved if the database is stored locally.

`Database` - Name of the database

`uid = ""` - Username

`pwd = ""` - Password

`Driver` - The structure of the remote database. I believe this should be set to `{SQL Server}`

- Note: Because there are so many different arguments, it may help to store the commands first as a character object using `paste()`, which allows structured code, allowing you to enter the arguments in the console over multiple lines. See **6.4. Worked Examples** below.
- Note: These arguments are from the DAT209x course slides, but they appear to be different from those in the ?help section.

`sqlQuery()` - Submit a query to an ODBC database and return the results. This is the basic function for querying SQL but function requires that you know some SQL.

- To perform certain simple querying, R has functions for some of the common tasks; these are sqlTables(), sqlFetch(), and sqlColumns().
- For extremely large tables, you will have to use `sqlQuery()`
- See 'SQL Syntax' below to see some basic SQL commands

`sqlTables()` - List tables on an SQL database that you are connected to

`sqlFetch()` - Read a table from an SQL database into a data.frame

`channel` - The object storing the open funnction (from odbcDriverConnect())

`sqtable` - The schema and name of the table to be fetched in the format: `"schema.name"`

`colnames = FALSE`

`rownames = TRUE`

- Be careful when fetching tables, as they may be *very large*. In these cases, you must use `sqlQuery()`

`sqlColumns()` - Query column structure of the SQL database (provides column names and 'type')

`close(connection)` - Close the connection to an ODBC database

6.3. SQL Syntax

SQL reads somewhat like English. I.e., the commands generally make sense when read in English; it also means that the order of the commands (appears to) matters.

`SELECT` - Selects a column in a particular table

`FROM` - Selects the table. This will be used in almost every command in conjunction with some condition

- `SELECT column1 FROM bi.data` is equal to `select(data, column1)` in `dplyr` and `data$column1` in R *(I think)*
- The "bi" in "bi.data" refers to the schema of the table. SQL tables must be read in with this info.

`WHERE` - Denotes that logic is about to be used. This is used in conjunction with logical operators to filter data. It works the same way in R, except R simply uses the logical operators without the 'where' statement.

- `SELECT column1 FROM bi.data WHERE column1 > 10` is equal to `data[data$column1 > 10]` in R

`COUNT` - counts something. Essentially replaces the function/argument `length`

`TOP n` - Displays the top 'n' values

`SELECT TOP 6 * FROM bi.data` is equal to `head(data)` in R

`AVG(), SUM(), STDEV()` - Equivalent to R functions

6.4. Worked Examples

6.4.1. DAT209x 7.2/7.3 example

"Connect to the following database and open the table named "manufacturer"

Server name: msedxus.database.windows.net
Database: DAT209x01
Login: RLogin
Password: P@ssw0rd"

Solution:

```
connStr <- paste(
  "Server=msedxeus.database.windows.net",
  "Database=DAT209x01",
  "uid=RLogin",
  "pwd=P@ssw0rd",
  "Driver={SQL Server}",
  sep = ";"
)
```

Note: Do not use spaces around '='

Note that the 'sep' argument is part of the paste() function to ensure that the character vector complies with the syntax of odbcDriverConnect()

```
conn <- odbcDriverConnect(connStr)
tab <- sqlTables(conn)
head(tab)  # this will display the top 6 tables. In this example we will pretend that 'manufacturer' is in the top 6
mf <- sqlFetch(conn, "bi.manufacturer")  # 'bi' is the schema of the table, found when you display head(tab)
```

Alternate solution (usin sqlQuery()):

```
query <- "
  SELECT Manufacturer
  FROM bi.manufacturer
"
sqlQuery(conn, query)
```

6.4.2. DAT209x lecture 7.4 example

"This is an example where fetching an entire table is infeasible because it is too large."

Count the number of rows within your table of interest:

```
sqlQuery(conn, "SELECT COUNT(*) FROM bi.salesFact")  # '*' indicates 'all'
1   1043986
```

Let's try to get some more information about the table:

```
sqlColumns(conn, "bi.salesFact")[c("COLUMN_NAME", "TYPE_NAME")]
```

This displays the names of the different columns and the "type" (equivalent to "class" in R)

```
    COLUMN_NAME       TYPE_NAME
1     ProductID          bigint
2          Date            date
3           Zip         varchar
4         Units             int
5       Revenue         numeric
```

Let's get some more information by displaying the first row:

```
sqlQuery(conn, "SELECT TOP 1 * FROM bi.salesFact")
    Product ID        Date          Zip    Units         Revenue
1         1    2012-10-20        30116         1         412.125
```

Let's fetch a subset of the data that we intend to work with.

```
df <- sqlQuery(conn, "SELECT * FROM bi.salesFact WHERE Zip = '30116' "
dim(df)
1   1000      5
```

We see that there are only 1000 rows, a size which R can deal with.

Note: when converted to R, SQL 'types' will be converted to R 'classes'. A quick way to check how your variables have been converted is to use `sapply(df, class)`

6.4.3. DAT209x exercise 7.1/7.2

"For the following tasks, use the following details for the SQL Server to connect to.

1) Server Name: msedxeus.database.windows.net
2) Database Name: DAT209x01
3) User ID: RLogin
4) Password: P@ssw0rd

1. Find the column names and types of the table "sentiment" from the SQL server, without fetching the entire table.
2. Find the number of rows in "sentiment" table without fetching the entire table."
3. Construct Construct a dataset based on the table "sentiment" from Exercise 7.1 with average score by Date for the State "WA". The dataset should also contain the column "Date".

Solution:

1.

```
connStr <- paste(
  "Server=msedxeus.database.windows.net",
  "Database=DAT209x01",
  "uid=RLogin",
  "pwd=P@ssw0rd",
  "Driver={SQL Server}",
  sep = ";"
)
conn <- odbcDriverConnect(connStr)
sqlColumns(conn, "bi.sentiment")[c("COLUMN_NAME", "TYPE_NAME")]
```

```
     COLUMN_NAME TYPE_NAME
1         DateID       int
2        StateID  smallint
3 ManufacturerID  smallint
4          Score   numeric
5   Manufacturer   varchar
6           Date  datetime
7          State   varchar
8            zip   varchar
9      ProductID    bigint
```

```
# 2.
sqlQuery(conn, "SELECT COUNT(*) FROM bi.sentiment")
```

```
1   21473
```

```
#3.
x1 <- sqlQuery(conn, "SELECT Date, AVG(Score)
FROM bi.sentiment
WHERE State = 'WA'
      GROUP BY Date"
)
options( digits = 4)
names(x1) <- c("Date", "Average Score")
head(x1)
```

```
        Date Average Score
1 2014-01-01         74.00
2 2014-02-01         71.56
3 2014-03-01         74.43
4 2014-04-01         71.33
5 2014-05-01         71.83
6 2014-06-01         70.42
```

7. Working with Text

Text mining is the way of the future! Analysis of Google searches, social media interactions, and extracting quantitative data from text sources (ex., prose, freeform questionnaires).

* Base R can perform some simple text analysis. However, for more advanced text analytics, look up the `tm` package.

* Even text analytics in base R are quite complicated. This is not a guide on how to work with text, but a basic introduction. You will likely have to read more in order to learn how to perform exactly what you're hoping to do.

7.1. Simple text manipulation in R (basic functions)

`grep()` - "Globally search a regular expression and print". Perform a text pattern search (the regular expression) and prints the index of the element that contains the text pattern.

`pattern` - A character string containing a regular expression to be matched in the given character vector. Supports vectors of length 1 (i.e., singles). If vectors with additional elements are used, only the first element will be used and a warning will be supplied.

`x` - The charater vector (or object that can be corerced into one) to be seached.

`ignore.case = FALSE` - Logical. By default, pattern matching is *case sensitive*

`fixed = FALSE` - Logical. If TRUE, `pattern` is a string to be matched 'as is'. Overrides all conflicting arguments.

`grepl()` - "Grep logical". Performs the same search as `grep()` but prints the output as a logical (ie., TRUE or FALSE) for every element of the object.

`gsub()` - "Grep substitute". Searches a text pattern and replaces all occurences.

`pattern`

`replacement` - The value that will replace the matched pattern. If possible, it will be coerced into a character string.

`x`

`sub()` - "substitute". Searches a text pattern and replaces the first occurence.

`replacement`

`strsplit()` - "String split." Splits a character vector into sub-strings depending on the arguments supplied.

`x` - Character vector.

`split` - Character vector containing a regular expression to use for splitting.

7.2. Constructing a regular expression

`*` - Indicates a break between arguments.

`.` - Indicates a sequence of arbitrary characters (possibly empty).

`[ ]` - Indicates a general search term where the number of occurences is unspecified. Ex.,

`[ :]` - Indicates a sequence of white space OR colons, without specifying how many to expect

`[0-9]` - Indicates a sequence of digits from 0-9, without specifying how many to expect

`x` - Replace this x with the character string(s) that you wish to search for

`( )` - Parentheses around certain arguments stores the data contained within them (the 'match'). The match is accessed through a 'back reference' " `\\1` "

7.3. Worked examples

7.3.1. DAT209x lecture example 1

We have messy data that look like this:

```
person.ID                       fruit
1   apple: 3 Orange : 9 banana:2
2   Orange:1 Apple: 3    banana: 10
3   banana: 3 Apple: 3  Orange : 04
```

- The data are stored as a character string, and cannot be easily co-erced into numerics
- The data labels and spacing between them are inconsistent
- However, they still follow a particular pattern that we can use a regular expression.

To find the number of oranges eaten by each person, we might write:

```
my.pattern <- ".*orange[ :]*([0-9]*).*"
```

In plain English this regular expression states, "Within the text, look for sequences that contain the word orange, followed by a colon and a number. Record the number."

More specifically, this regular expression states, "The pattern includes some arbitrary characters; followed by the string 'orange'; followed by an unspecified number of colons AND/OR whitespace; followed by an unspecified number of digits ranging from 0-9, which should be stored; followed by some arbitrary characters."

```
sub(my.pattern, \\1, df$fruit, ignore.case = TRUE)
[1] "9""1"    "04"
```

This function states, "Search `df$fruit` for terms that meet the conditions of the expression `my.pattern`. Instead of printing these terms back, substitute the term with the data stored by `my.pattern` and print that back. While doing this, ignore case."

Note that our output is a character string. We can now convert this to a number using `as.numeric()`

7.3.2. DAT209x Exercise 8.3

'Given the text string:

```
my.text <- "Over the last decade, bluetongue virus have spread northwards from the
mediterranean area. Initially this was ascribed to climate changes, but it has since
```

```
been realized that a major contributing factor has been new transmitting vectors,
culicoides obsoletus and culicoides pulicaris, which have the ability to aquire and
transmit the disease. Recently, schmallenberg virus has emerged in northern europe,
transmitted by biting midges as well."
```

Modify the object my.text so that the words bluetongue, culicoides, europe, mediterranean, northern and schmallenberg all start with a capital letter'

Solution:

We will use `gsub()` rather than sub, since we want to replace all instances of the words, and there are multiple instances of the word 'culicoides.' `gsub()` does not support the use of vectors, so we will have to apply a `for()` loop:

```
search.term <- c("bluetongue", "culicoides", "europe", "mediterranean", "northern",
"schmallenberg")
replace.term <- c("Bluetongue", "Culicoides", "Europe", "Mediterranean", "Northern",
"Schmallenberg")

my.new.text <- my.text
for(i in 1:6)my.new.text<-gsub(search.term[i],replace.term[i], my.new.text)
my.new.text
[1] "Over the last decade, Bluetongue virus have spread northwards from the Mediterranean area. Initiall
y this was ascribed to climate changes, but it has since been realized that a major contributing factor
has been new transmitting vectors, Culicoides obsoletus and Culicoides pulicaris, which have the ability
to aquire and transmit the disease. Recently, Schmallenberg virus has emerged in Northern Europe, transm
itted by biting midges as well."
```

7.4. Working with Dates

Dates often cause problems when performing import/export, as different software will store date information differently.

In R, dates are character strings with special properties. For example, arithmetic functions can be applied to dates. R has two classes to represent date and time objects, and two functions to construct them:

`as.Date()` - Contains calendar date information

- `x` - The object to be converted. It can be a character string that specifies the date following `format` or it can be a numerical vector, provided that there is a character string that specifies the date in `origin`
- `format` - What format is the date given in? By default it will first attempt "Y-m-d", then "Y/m/d". It will call an error if neither works.
- `origin` - character string of length 1. This represents the 'start date'
- Note: The scale for `as.Date()` is days

`as.POSIXct()` - Contains date *and time* information (i.e., it is a more precise version of `as.Date()`).

- Note: The scale for `as.POSIXct()` is seconds

`julian()` - Returns the number of calendar days between each element of '`x`' and `origin`.

- `x` - Vector. Accepts dates and POSIXt as inputs

`origin` - Start date. Unlike `as.Date()` and `as.POSIXct()`, this date does not take on a default format. Therefore, it must be defined as either a date or POSIXt using `as.Date()` or `as.POSIXct()`.

8. Simulation

8.1. Random numbers

Random numbers in R are pseudo-random numbers. The default pseudo-random number generator in R is the *Mersenne Twister* which has a period of $2^{19937}-1$. This exceeds the number atoms in the universe by over 5000-fold.

- For all practical purposes pseudo-random numbers in R behave like true random numbers.
- There is an additional benefit, in that pseudo-random numbers can be generated reproducibly by setting the seed.

8.2. Simulating complex systems

Simulation is less efficient and less accurate than calculating. For example, the example given in **8.3.1. Performing integration of a function** can be calculated more quickly and more accurately using known rules about integration. However, simulation can be used when it is not possible to calculate, or too complex to do so.

There are three main steps to applying a stochastic simulation model:

1) Initialization
    - Identify variables that need to be tracked during the simulation
    - Define the starting values for these variables
    - Save the code as a .txt
2) Loop step
    - Each step must advance the loop forward by a step and update the value of the variables
    - Save the code as a .txt
3) The actual simulation
    - Create a loop that will source step (1) and (2).
    - The loop must include the number of trials and store variables of interest

8.3. Interesting examples of simulation

8.3.1. Performing integration of a function

Consider the function $f(x) = e^{2\cos(x-\pi)}$

Calculate the integral

$$\int_0^{2\pi} f(x)dx$$

We can estimate this using a Monte Carlo simulation:

- We can simulate 'n' uniformly distributed random points (x_n, y_n) over the interval $(0, 2\pi) \times (0, 8)$
    - We select 8 here because it is above the maximum value of the function, but including a larger range does not really affect the simulation.
- We then count the frequency of points falling within the area under the curve given by the function via:
  $\hat{P}_u = \frac{points\ below}{total\ points}$
- Multiply the frequency $\hat{P}_u$ by the total area simulated to estimate the integral
  $\int_0^{2\pi} f(x)dx \approx 16\pi \times \hat{P}_u$

```
set.seed(132423)
x <- runif(1000000, min = 0, max = 2*pi)
f.x <- exp(2*cos(x-pi)
y <- runif(1000000, min = 0, max = 8)
phat.under <- sum(y < f.x) / length(y)
[1] 14.32755
```

This problem can also be solved using R's built-in `integrate()` function:

```
f.x2 <- function(x) exp(2*cos(x - pi))
integrate(f.x2, lower = 0, upper = 2*pi)$value
[1] 14.32306
```

Standard uncertainty for this Monte Carlo integration has been shown to be +/- 0.05

8.3.2. Simulation of two-step dice rolls

"Consider the following set up: You roll two dice, and count the number x of eyes on the dice. You then roll a number of dice corresponding to x, and record the number y of eyes on the dice. Simulate the process 1.000 times, and make a histogram of t he values of y."

Solution A:

```
x <- 0
y <- matrix(0, nrow = 100, ncol = 10)
for(i in 1:1000){
  x    <- sum(sample(1:6, size = 2, replace = TRUE))
  y[i] <- sum(sample(1:6, size = x, replace = TRUE))
}
hist(y)
```

Solution B:

```
dice.roll2 <- function(n){
  d <- sum(sample(1:n, size = 2, replace = TRUE))
  e <- sum(sample(1:n, size = d, replace = TRUE))
```

```
return(e)
}
p <- replicate(dice.roll2(6), n = 1000)
hist(p)
```

8.4. Checking run-times

Fill this in

9. Statistical Models

A linear model in statistics simply describes a variable (Y) that can be described in terms of another (X) using a straight line. $\in$ is a term which describes error (i.e., how much does X deviate from its systematic dependence of Y)

$$Y_i = \alpha + \beta X_i + \epsilon_i$$

The residuals (ϵ_i) should be:

- stochastically independent
- identically and nearly-normally distributed
- with mean = 0, variance = σ^2

9.1. `lm()` and `glm()`

9.1.1. `lm()`

`lm()` is a powerful function that fits a general linear model to the data.

- The model does not necessarily need to be linear - only the co-efficients. Many non-linear functions (such as polynomials or exponentials) can easily be modeled using `lm()` as well. A simple example is presented in Section Error! Reference source not found..**2.** Error! Reference source not found.
- Note: Using `lm()` appropriately requires that the assumptions for a general linear model are met. Review the Linear Regression section in the Statistics Manual to see if a general linear model can be applied.

`lm()`

`formula` - R object that specifies the linear model.

- Note that the formula can be created and saved as an object using the `formula()` function. This allows you to manipulate the model through the formula object without having to invoke the `lm()` function. This apparently saves processing time.
- See Section **1.3. Formula** if you are having trouble expressing the desired formula.

`data` - Data frame or list containing the variables found in `formula`

`subset` - An optional vector specifying which subset of observations should be used

`na.action` - How should NA values be handled? By default, they are ignored.

The object is a list of length 12 containing information about the model. Ex.,

```
names(m1)
[1] "coefficients"  "residuals"     "effects"
[4] "rank"          "fitted.values" "assign"
[7] "qr"            "df.residual"   "xlevels"
[10] "call"          "terms"         "model"
```

9.1.1.1. Graphical diagnostics for residuals

`plot(m1)` can be used to display 4 diagnostic graphs for the model. Additional diagnostic plots can also be called. See `?plot.lm`.

It is recommended to first apply `par(mfrow = c(2,2))`.

- Residuals vs. Fitted
- Normal Q-Q for the Residuals
- $\sqrt{\text{Standardized residuals}}$ vs. Fitted
- Standardized residuals vs. Leverage

For all practical purposes, the `plot()` function discussed above will work. However, the graphs can also be constructed interactively for better overview. This constructs:

- A scatter plot of Y ~ X, along with the best-fit line (i.e, the model)
- Scatterplot of Residuals vs. Index
- Histogram of Residuals
- Q-Q plot for the Residuals

```
m1 <- lm(y ~ x)
sim.residuals <- function(x) dnorm(x, mean = 0, sd = sd(residuals(m1)))
    ## Note the estimation here that the variance of the residuals is equal to the
variance in the error terms.

par(mfrow = c(2,2))
plot(y~x); abline(m1)  # Scatterplot
plot(residuals(m1)); abline( h = 0, lty = 3)  # Residual plot
hist(residuals(m1), prob = T); curve(sim.residuals, add = T, lwd = 3)  # Histogram
qqnorm(residuals(m1)); qqline(residuals(m1))  # QQ plot
```

9.1.2. `glm()`

`glm()`, 'Generalized Linear Model'

- A *Generalized* linear model (as in `glm()`) is a more generalized general linear model (as in `lm()`), and therefore encompasses `lm()`. Examples of distributions that can be modeled with glm(): **Poisson, binomial, multinomial, normal, gamma.**
- Usage of `glm()` is similar to `lm()` with the exception of the `family` argument, which specifies which distribution and link function should be used to compute the model.

`glm()` - Error! Reference source not found.**.** Error! Reference source not found. provides a good example on how to use `glm()`

`formula` - R object that specifies the model.
- For logistic regression, the response variable can be given as case-by-case data (a list of 0s and 1s) or as aggregated data, as a 2-column matrix with failures in column 1 and successes in column 2.

`family` - String. A description of the error distribution and the link funciton to be used in the model. I.e.,
"`gaussian`" links to "`identity`", "`log`", "`inverse`"
"`binomial`" links to "`logit`", "`probit`", "`cauchit`", "`log`", "`cloglog`"
(see R help for other families and their respective link functions)

`data` - Data frame or list containing the variables found in `formula`

`subset` - An optional vector specifying which subset of observations should be used

`na.action` - How should NA values be handled? By default, they are ignored.

9.1.3. Extraction functions

A number of extraction functions can be applied to `lm()` and `glm()` to extract additional information from the model. This is list of some of the most useful, but there are many more. Many of these functions appear redundant because they are built-into the `lm()` object, but they can also be applied to other fitted model objects.

`anova()` - ANOVA test table for model reduction. Tests which variables account for the most variance.

`object` - fitted model object

`test` - Character string. Which test statistic should be used to test the model?
- Possible inputs: `"F", "Chisq", "LRT", "Rao"`

`drop1()` - Test the effect of dropping each individual term from the model. You may remove the 'least' significant' term and repeat the function step-wise to improve the model. `drop1()` will only test to drop terms that are not part of an interaction term. Note that this differs from anova in that

`object` - A fitted model object

`test` - Character string. Which test statistic should be used to test the model?
- Possible inputs: `"F", "Chisq", "LRT", "Rao"`
- Note: `"F"` should be used for `lm()`

- For usage, see Error! Reference source not found.**.** Error! Reference source not found..

`coef()` - Returns the estimated model of parameters.

`object` - `lm` or other model object. The model you just made.

`cofint()` - Confidence intervals for estimated model parameters.

`object` - `lm` or other model object. The model you just made.

`hatvalues()` - Calculates leverage statistics for the variables in a model.

`model.matrix()` - The design matrix (*I'm not sure what this does*).

`object` - `lm` or other model object. The model you just made.

`predict()` - Returns predicted values from the model. Can also be applied to a new data set to make predictions using the same model.

`object` - `lm` or other model object. The model you just made.

`newdata` - Optional data frame. Data set used to make new predictions based on the model in `object`. If omitted, fitted values are used.

`se.fit` - logical. Default = FALSE. If TRUE, the output of `predict()` becomes a list containing the predicted values, '`fit`,' and their standard errors, '`se.fit`.' Thus, `se.fit` can be used to construct confidence intervals.

`interval` - string. Type of interval to calculate: "`none`", "`confidence`", "`prediction`".

`level` - numeric. Confidence level (ex., 0.95).

`residuals()` - Returns the residuals

`object` - `lm` or other model object. The model you just made.

`rstandard()` - Returns the studentized (standardized) residuls

`object` - `lm` or other model object. The model you just made.

`summary()` - A summary printout, and access to summary statistics. This summary is a list object itself, with sub-elements that can be accessed. Ex.,

```
names(summary(m1))
 [1] "call"          "terms"         "residuals"
 [4] "coefficients"  "aliased"       "sigma"
 [7] "df"            "r.squared"     "adj.r.squared"
[10] "fstatistic"    "cov.unscaled"
```

`update()` - Allows you to refit a model.

`object` - The model object to be updated. This function can be used with only this argument (Ex., if the formula is to remain the same, but the data frame used to construct the model has changed)

`formula` - The new formula to be used to construct the model.

- ' `~.` ' indicates 'the same formula as before'
- Terms may be added or removed terms using the '+' and '−' operators

- Note: in many cases it is better practice to simply create a new object rather than replace the existing one.

`vcov()` - Covariance matrix for estimated model parameters

`vif()` - Calculates the *variance inflation factor* for each variable in the model. Not included in base R, but is part of the `cars` library.

`mod` - `lm` or other model object. The model you just made.

`which.max()` - related to `hatvalues()`; identifies the index of the observation with the largest leverage.

9.2. `lda()` and `qda()`: Linear and quadratic discriminant analysis:

LDA and QDA models can be fit using the `lda()` and `qda()` functions as part of the `MASS` library. The syntax for `lda()` is identical to that of `lm()`.

`lda()` - Performs linear discriminant analysis

- `formula` - R object that specifies the linear model.
- `data` - Data frame or list containing the variables found in `formula`
- `subset` - An optional vector specifying which subset of observations should be used
- `na.action` - How should NA values be handled? By default, the procedure will fail.

** `qda()` works the exact same way as `lda()`.

Note: the `plot()` function can be applied to `lda` objects.

9.2.1. Evaluating `lda()` classification error rates

Confusion matrices

After using `predict()` to create a data frame containing a vector of predictions using the classification model, you can generate a confusion matrix with simple commands:

```
lda.prediction <- predict(lda.model, new.data)   # Make predictions using the lda model (response variable must have the same name as in the formula)
lda.classification <- lda.prediction$class       # Extract the predicted classifications
table(prediction = lda.classification, truth = test.data$classification)      # Generate a confusion matrix that compares your predicted classifications to the true classification of the test data set.
```

```
          truth
prediction   0   1
         0 489  87
         1  17 407
```

The mean is equal to 1 – the overall error rate

```
mean(lda.classification == test.data$classification)
 [1] 0.896
```

A function for which displays the confusion matrix and prints common classification terms.

```
confusion.matrix <- function(yhat, y) {
       #  Inputs must be vectors of 0 and 1 outcomes (and they should be equal length)
    print("Confusion matrix:")
    print(table(prediction = yhat, truth = y))
    print(paste("Sensitivity:", sum(yhat == 1 & yhat == y) / sum(y == 1)))
    print(paste("Specificity:", sum(yhat == 0 & yhat == y) / sum(y == 0)))
    print(paste("PPV:", sum(yhat == 1 & yhat == y) / sum(yhat == 1)))
    print(paste("NPV:", sum(yhat == 0 & yhat == y) / sum(yhat == 0)))
}
```

Adjusting prediction thresholds

By default, predictions are made by assigning observations to the class for which they have the highest posterior probability. However, you can change this threshold manually to generate your own predictions. In this example, we change the threshold to classify a '1' from 0.5 to 0.4.

```
sum(lda.prediction$posterior[, 1] >= 0.40)
[1] 602

sum(lda.prediction$posterior[, 1] < 0.6)
[1] 448
```

This example uses sums to give us an indication of the overall error rate, but we could just as easily create a new vector of predictions using this rule.

9.3. knn()

`knn()` is part of the `FNN` library, though there are many other `knn` packages.

`knn()` - Outputs a vector of factors which are estimates of *y* using the *k* nearest neighbours.

- `train` - data frame. Data frame containing all desired predictors, but NOT the response variable. All variables must be numeric.
- `cl` - vector. The vector of true classifications in the training data.
- `test` - data frame. A second data frame with the same number and name of columns as in the `train` set.
- `k` - Integer. The number of nearest neighbours to use
- `prob` - logical. Default = FALSE. If TRUE, it will display the posterior probabilities (i.e., the proportion of neighbours that voted for the chosen outcome).
- `algorithm` - String. Which algorithm should be used to find the knn: `"kd_tree"`, `"cover_tree"`, `"brute"`

Notes:

- `knn()` can also be used for regression if *y* is continuous, though it can only take on values which already exist in the data set.
- Decisions are made by majority vote, with ties being broken at random. For reproducibility, setting a seed may be a good idea.
- Does not accept NAs.
- The knn model does not know how to properly handle variables with different units (i.e., a difference of $1000 will be weighed more than a difference of $1K). To correct this, apply `scale()` to normalize the data by mean and sd.

Example:

```
library(ISLR); library(dplyr)
set.seed(1)

knn.train.data <- filter(Smarket, set == "train") %>%
    select(Lag1, Lag2, Direction)

knn.test.data <- filter(Smarket, set == "test") %>%
    select(Lag1, Lag2, Direction)

knn.result <- knn(train = knn.train.data[, -3],
                  cl = knn.train.data[, 3],
                  test = knn.test.data[, -3],
                  k = 1)

table(prediction = knn.result, truth = knn.test.data[, 3])
```

```
          truth
prediction Down Up
      Down   43 58
      Up     68 83
```

9.3.1. Assessing the knn model

Identify the *k* nearest neighbours selected

```
indices <- attr(knn.result, "nn.index")
indices[1:20]
```

Identify which group the *k* nearest neighbours fall into

`knn.checkclass <- matrix(train.data$classification[indices], ncol = k)` # Finds the classification of the *k* nearest neighbours from the training data. `ncol = k` should be the same `k` as in the original knn model.

I'm not sure what this does:

```
attr(knn.result, "nn.dist")
```

Example - Logistic Regression by 2 variables.R'

Applications of `glm()` - Classification Models and ROC Analysis.

Too complicated for me right now... Review DAT209 exercise 11.4.

10.Graphing in R

10.1. Plotting functions

`pairs()` - Plots a matrix of pairwise scatterplots for a data frame. For high level overview of a large number of variables, try the `corrplot` package (**11.2.1. `corrplot` to visualize pair-wise correlations**).

- x - A data frame or list of vectors. Use the name of the data frame to view all pairwise scatterplots. Otherwise, using the formula method is easier.
- `formula` - A formula such as `~ x + y + z`. Each term should be a numeric vector. Response variables are interpreted as any other variable, so the formula leads with a `~`.
- data - A data frame or list from which the variables in `formula` should be taken.

`plot()`

- `border` - string or integer. Colours the border.
- `main` - String. Main title.
- `sub` - String. Subtitle.
- `type` - String. What type of plot should be drawn?
  - `"p"` for points
  - `"l"` for lines
  - `"b"` for both
  - `"c"` for the lines part of `"b"`
  - `"s"` for stair steps
  - `"S"` for other steps
  - `"o"` for both 'overplotted'
  - `"h"` for 'histogram'-like
  - `"n"` for no plotting
- `xlab/ylab` - String. Sets axis titles.
- `xlim/ylim` - Numeric of length 2. Sets the min and max axis limits. Takes on the form `c(n1, n2)`

10.2. Graphical Parameters

10.2.1. Plotting arguments

10.2.2. `par()`

Many of the arguments available to the par() function can be used as arguments in plotting functions.

`par()`

`col` - 'Colour.' Integer vector (or coerceable into an integer), or a character string. Assigns colour to the data points. See http://www.stat.columbia.edu/~tzheng/files/Rcolor.pdf

`las` - 'Axis label style.' Integer. Sets the orientation of axis labels. Default = 0

0: parallel to axis 1: horizontal 2: perpendicular to axis 3: vertical

`lend` - 'Line end style.' Integer or string. Default = 0

0: `"round"` 1: `"butt"` 2: `"square"`

`lty` - 'Line type.' Integer or string. Default = 1

0: `"blank"` 1: `"solid"` 2: `"dashed"` 3: `"dotted"`

4: `"dotdash"` 5: `"longdash"` 6: `"twodash"`

`lwd` - 'Line width.' Positive numeric. Default = 1. *Note*: Interpretation is device-specific.

`mfrow/mfcol` - Numeric vector of length 2. Specifies the number of rows and columns of plots should appear in the plotting page

`pch` - 'Point character.' Integer vector (or coerceable into an integer), or a character string. Assigns a character to represent each point.

- Useful integers: 20: • 150: - 151: –
- Useful character strings: . + - | *

* *Note*: Factors are coerceable into integers. Therefore, a variable of mode 'factor' in a data frame can be used in many of these arguments (ex., `col`, `pch`) to visualize different sub-groups on a plot.

10.3. Lines

`abline()` - Plots a single line connecting the coordinates supplied.

`curve()`

`grid()` - Low-level plotting function that adds a grid on the current plot.

`nx, ny` - The number of cells of the grid in the x and y direction. Default = NULL, meaning that gridlines will alinfe with the major tickmarks of the plot. When NA, no gridlines will be drawn for that direction.

- Can also be customized using `col`, `lty`, and `lwd`

`legend()` - A low-level graphing function that adds a legend to the current plot. Run the code in Error! Reference source not found.**.** Error! Reference source not found. to see a good example of `legend()`.

`x, y` - x and y coordinates on the current plot that the top-left corner of the legend should be positioned.

`keyword` - Character string. Ex., `"top left"`

`legend` - Character or expression vector to appear in the legend.

`col` - Integer vector. Which colours are used in the current plot.

`lty, lwd` - Integer or string. Specify the line type or width. At least one of these must be specified to make a line in the legend.

`pch` - Integer. What point type is used in the current plot (default is 1)

`cex` - 'Character expansion factor'. Numeric. Determines how large the legend text is. Default = 1

- Note: The size of the legend is relative to the size of the plot. The plot area should be blown up to full size before calling `legend()`.
  - You can get around this using `dev.new()` to make a new windowed plot with the desired dimensions.
- It is important to check that the parameters match the original plot, as R does not cross-reference the two to check for you. An example of how you can avoid this is by directly referencing the data frame.

```
legend(0, 80, unique(x$SEX), col = x$SEX, pch = 1)
```

`lines()` - A low-level graphing function that takes coordinates and joins them with line segments.

`x, y` - coordinate vectors of points to join

- If only two coordinates are used, this is essentially the same as `abline()`
- A linear equation can be created by entering $mx + b$ in the '`y`' argument. Ex.,

```
x.val <- seq(0,100)
lines(x = x.val, y = (3*x.val + 12.4))
```

`type` - 1-character string giving the type of plot desired. See `plot()` for full details.

`...` -

- The coordinates can contain NA values. If a point contains NA in either its x or y value, it is omitted from the plot, and lines are not drawn to or from such points. Thus *missing values can be used to achieve breaks in lines.*
- To prevent spider-webbing of lines, it may help to order your data in ascending order. See **2.3.4. Ordering data in a data frame**.

`rug()` - Adds ticks next to the axis showing the value of points.

10.4. ggplot2

http://docs.ggplot2.org/current/

+ Allows a shorter syntax for creating custom plots compared to base R.
+ Plots are stored as R objects, allowing graphics to be built/modified incrementally
- Allows less customization and handles 3D graphics poorly compared to and base R
- Graphics take a long time to generate

`qplot()` - A shortcut wrapper for creating graphics similar to base `plot()`.

`x, y, ...` - Vector. variables to be graphed along the X and Y axes

`data` - Data frame that the data are from (optional). If unspecified, it will create one.

`facets` -

`margins` -

`geom` - Character vector. Specifies which geom(s) should be drawn. Default = "point" if x and y are s specified, and "histogram" if only x is specified.

`xlim, ylim` - Numeric vector (length = 2). X and Y axis limits.

`log` - Character. Which variables should be log transformed? ("`x`", "`y`", "`xy`")

`main` - Character. Main title.

`xlab, ylab` - Character. Axis labels.

`asp` - Numeric. The Y / X aspect ratio.

`+` - Operator used to denote a new layer on top of an existing `ggplot` object.

`ggplot()` - A 'gg plot' object. Arguments initially passed are not enough to construct a plot. Instead, arguments must be layered on sequentially.

`data` - The default data set to use for the plot. If not already in a data frame, it will be converted to one.

`mapping` - The default variables to plot within the `aes()` function.

- Note that one or both of these arguments may be missing. These arguments should only be specified if the same data frame and variables will be used for the entire plot. If different ones will be used, they should be left unspecified and added later.

- Additional parameters can be layered on using the below function stems. An exhaustive list of all the possible functions is available at: http://docs.ggplot2.org/current/.

  * Examples of each of these functions can be accessed using the `example()` function.

  - Geoms - Short for 'geometric objects.' Describes the type of plot you will produce.
  - Scales - Control the mapping between data and aesthetics.
  - Coordinates - Adjust the mapping from coordinates to the 2d plane of the computer screen
  - Faceting - Display subsets of the data in different panels
  - Position adjust - Fine tune positioning of objects to get effects like dodging/jittering
  - Annotation - Functions for adding annotations to a plot
  - Themes - Control non-data components of the plot
  - Aesthetic - Define aesthetic mappings
  - Other

11. R Packages

Useful R packages that are widely accepted:

1) Rcpp - Integration of C++ into R code
2) ggplot2 - grammer of graphics - plotting tools
3) stringr - Easy to learn tools for regular expressions and character strings.
4) plyr - tools for splitting / manipulating data
5) foreign - for reading in data from SAS or SPSS
6) dplyr - Tools for subsetting / manipulating data

11.1. Data manipulation in `dplyr` (basic functions)

'dplyr' is a package that helps with subsetting tables (i.e., data frames) in a way that is closer to English than native 'R' by performing a given function using a single verb.

dplyr also provides improved consistency over base R:

- The first argument is always a data frame.
- Subsequent arguments describe what to do with that data frame
- The output is always a data frame

* denotes that the function is faster in `dplyr` versus the equivalent code in R (so use when possible). These include:

- `filter()` to subset rows (vs. a combination of `[]` and `$` operators)
- `arrange()` to order data frames (vs. `order()`)
- `group_by()` to aggregate data (vs. `aggregate()` or `tapply()`)
- `mutate()` to create new columns (vs. the `$` operator)

`%>%` - pipe operator. Place this symbol between two functions; this will tell the program to use the output of the first function as the input for the second. This allows you to perform multiple data manipulations in one line.

`*arrange()` - Re-arranges rows (default is ascending order). Subsequent arguments are used to arrange the rows in the event of a tie in the preceding column.

- Ex., `arrange(x, column1, desc(column2))` arranges rows in ascending order based on column1, followed by descending order based on column2.
    - In base R, the equivalent code is: `x[order(x$column1, x$column2), ]`

`*filter()` - Used to subset rows from a data.frame. The first arugment is the name of the data frame; subsequent arguments are the expressions that you wish to filter. Works in a similar way to `subset()` in base R.

- Ex., `filter(x, age == 3, eyes == brown) will` keep rows in which age column is 3 and the eyes column is brown.
    - In base R, the equivalent code is: `x[x$age == 3 & x$eyes == brown, ]`
- Note: When using `filter()`, in `dplyr` observations will retain the same row numbers as the original data frame. When using `subset()`, the observations will be re-numbered (starting from 0). All of the values are the exact same, but the row number should not be used to cross reference these data.

`*group_by()` - Groups data by one or more grouping variables. similar to `aggregate()` or `tapply()`

`...` - Variable name. The variable(s) from the data frame that is used for grouping.

`.data` - Data frame. The object from which the variables are taken

`add` - Logical. Should the function override or add to existing groups? Default = FALSE

`*mutate()` - Creates a new column in your data frame from an existing column. You can create muliple columns using a single command. This is similar to transform() in base R, but mutate() can refer to columns that you've created within the same command.

- Ex., `mutate(x, BMI = (mass / (height)^2) )` will create a new column named 'BMI' based off of an equation using the 'mass' and 'height' columns.

`rename()` - Allows you to rename columns.

- Ex., `rename(x, eyes = eye_colour)` will rename the 'eyes' column to 'eye_colour'

`select()` - Create a new data.frame using only the selected columns, allowing you to focus only on the variable(s) of interest. There are a number of helper functions that can be used as arguments for select() such as starts_with(), ends_with(), matches(), contains(). See ?select for more details.

- Ex., `select(x, age, eyes)` will create a new data.frame with the columns age and eyes.
- select() is actually slower at subsetting columns than base R

`slice()` - Like `filter()`, also subsets rows, but does so based on position rather than value in a certain variable.

`summarise()` - Summarises all of the values in a column to a single value. Commonly used in conjunction with `group_by()` to achieve a similar effect to `aggregate()`.

`.data` - A `tbl` or data frame.

... - Summary functions to evaluate, with the variable of interest within them. Ex., `sum(Sales)`

`transmute()` - Works the same way as mutate() but replaces the old variable(s) with the new one.

`unlist()` - Turns a lists, such as a column in a data.frame, into a numeric vector. This allows us to perform other functions on the data.

11.1.1. Examples of data manipulations using `dplyr`

Extracting a single variable from a dataframe for analysis:
In this example, we want to know the distribution of ages for individuals from Canada. We have already loaded the data.frame, 'x'.

```
library(dplyr)
age_CAN <- filter(x, country == "Canada") %>% select(age) %>% unlist
```

```
# The equivalent code in base R is:
age_CAN <- x[(x$country == "Canada"), colnames(x) == "age", ]
# From here, the data are stored as a numeric vector and can be analyzed using most mathematical functions in R.
```

11.2. Data Visualization packages

11.2.1. `corrplot` to visualize pair-wise correlations

`corrplot()` - Produces a matrix of plots with each pairwise correlation.

`corr` - Matrix. The matrix of correlations to plot. Must be generated with the `cor()` function.

`method` - Specific character. The visualization method to be used:

`"Circle"` (default) `"square"` `"ellipse"`

`"number"` `"pie"` `"shade"` `"color"`

`type` - Specific character. Which portion of the correlation matrix should be displayed?

`"full"` (default) `"upper"` `"lower"`

`add` - Logical. If TRUE, will add to the current plot. Default = FALSE.

`col` - Integer or character vector. The colour of the glyphs.

`bg` - Integer or character. The background colour.

`title` - Character. Title.

`order` - Specifc character. How to order the variables on the corr plot.

"`original`" for the original order of the data (default)

"`AOE`" for angular order of eigenvectors

"`FPC`" for first prncipal component order

"hclust" for hierarchical clustering order. Allows use of addrect argument.

"alphabet"

- Note: If using add = TRUE to display the corrplot in two different ways, make sure that both plots are ordered the same way.

addrect - Integer. Number of rectangles to draw on the graph according to the hiericharl cluster. Only valid when order = "hclust"

p.mat - Matrix of *p*-values for each pair-wise regression. Get this from the test of your choice. Most commonly cor.test().

sig.level - Numeric. Cut-off significance level.

insig - Specific character. How should "insignificant" correlations be represented on the corrplot?

"pch" (default) "p-value" "blank" "n"

11.3. Decision tree packages

11.3.1. rpart to model data using decision trees

'rpart' stands for "Recursive Partitioning." Recursive partitioning is a tool that allows us to develop easy to visualize decision rules for predicting outcomes by Classification or Regression; this is often called the **C**lassification **A**nd **R**egression **T**ree (CART) method.

A useful guide on using rpart can be read here: Reference\rpart.pdf

Limitations

- Decision trees assign priority to variables that explain most of the variance; therefore, they may miss out on other explanatory variables.
  - A simple example is a decision tree which dispenses change: We must dispense 30c in a world where no 5c coins exist.
    - The tree would see that a 25c coin explains most of the variance, and thus perform:

      25 + 1 + 1 + 1 + 1 + 1
    - However, in reality a faster way to dispense this would be:

      10 + 10 + 10
- Again because of the way these trees assign priority, they are biased toward factors with a large number of levels. This is because they so many possible ways of splitting the data that they can do so in ways that perfectly explain the variance, but have no basis in reality.
- Decision trees are prone to over-fitting.

Generating a model

rpart()

formula - Formula object. Same as other models.

data - Data frame from which the formula variables are taken from.

`method`- Specific string. arguments can be used, but "`class`" and "`anova`" are used commonly to do classification and regression, respectively.

`control` - Optional parameters for controlling tree growth, fed into the `rpart.control()` function. Read the help page.

- Ex., `control = rpart.control(minsplit = 30, cp = 0.001)` requires that the minimum number of observations in a node be 30 before attempting a split and that a split must decrease the overall lack of fit by a factor of 0.001 (cost complexity factor) before being attempted.

`rpart.control()`

`minsplit` - Integer. Minimum number of observations that must exist in a node in order for a split to be attempted. Default = 20.

`minbucket` - Integer. Minimum number of observations that must exist in a terminal node. If only one of `minsplit` or `minbucket` is specified, `minbucket / 3 = minsplit` is automatically specified.

`cp` - "Complexity parameter." Numeric. Any split that does not decrease the overall lack of fit by a factor of `cp` will be deemed unimportant *a priori* and will not be computed. Default = 0.01

- Ex., for `anova` splitting, `cp` takes on a tangible value; each step must decrease R^2 by `cp`.

`maxcompete` - Integer. The number of competitor splits retained in the output. Instead of just informing the user which split was chosen, it may be helpful to know which variable came in second, third, etc. Default = 4.

`maxsurrogate` - Integer. The number of surrogate splits retained in the output. If set to 0, computation time will be ~1/2. Default = 5.

`usesurrogate` - Integer from 0:2. How to use surrogates in the splitting process.

- `0`: Display only; an observation with a missing primary split variable will terminate at that node. Corresponds to the action of `tree`.
- `1`: Use surrogates to split subjects if the primary variable is missing. If the surrogate is missing, the observation will terminate at that node.
- `2`: Use surrogates as in 1. If all surrogates are missing, then send the observation in the majority direction (Default). Corresponds to the recommendations of Breiman *et. al* (1984).

`xval` - Integer. Number of cross validations. Default = 10.

`surrogatestyle` - Integer from 0:1. Controls the selection of the best surrogate.

- `0`: Use the total number of correct classification for a potential surrogate variable (Default). This option more severely penalizes covariates with a large number of missing values.
- `1`: Use the 'percent correct' classifications over the non-missing values of the surrogate.

`maxdepth` - Integer. The maximum depth of the tree (the root counts as depth = 0). Default = 30.

Viewing results

`predict()` - Just like in base R, returns a vector of the predicted responses from a fitted `rpart` object.

`object` - `rpart` object. The model that you wish to extract the predictions from.

`newdata` - Data frame. The "prediction/test" data frame, which contains all of the variables included in the original `rpart` model.

`type` - String denoting the type of predicted value to be returned.

- "`prob`" returns a matrix of probabilities of each outcome of the original outcome variable.
- "`class`" returns a classification (i.e., binary decision).
- "`vector`" - returns a vector of the predicted response. For regression, this is the mean response at the node; for Poisson trees it is the estimated response rate. For classification trees it is the predicted class.

`plotcp()` - Plot cross-validation results

`post()` - Create a postscript plot of the decision tree.

`print()` - Print a text version of the tree. Allows overview in the event of overplotting.

`printcp()` - Display cp table

`prp()` - Plot the `rpart` object. Also allows pruning. See below under "Extracting results"

`fancyRpartPlot()` - Plots a nicer version of the decision tree. Requires the packages `rattle`, and `rpart.plot`

`rsq.part()` - Plot approximate R^2 and relative error for different splits. Labels are only appropriate for the "anova"

`summary()` - Provides a detailed report of the model, including surrogate splits

Pruning results

When selecting the number of splits to use in a tree, the convention is either:

(A) use the best tree possible (i.e., lowest cross-validation erorr)

(B) the simplest tree within one standard error of the best tree (i.e., the '1 - SE' method).

Both of these can be seen using the `printcp()` command. Method (A) provides the best fit, but Method (B) provides some protection against overfitting by using a simpler tree.

Ex.,

```
            CP nsplit rel error    xerror       xstd
1  0.4444444444      0 1.0000000 1.0000000 0.04244576
2  0.0307017544      1 0.5555556 0.5555556 0.03574957
3  0.0233918129      3 0.4941520 0.5233918 0.03497048
4  0.0204678363      4 0.4707602 0.5000000 0.03437157
5  0.0102339181      5 0.4502924 0.5233918 0.03497048
6  0.0065789474      8 0.4181287 0.4970760 0.03429471
7  0.0058479532     14 0.3771930 0.5116959 0.03467453
8  0.0043859649     16 0.3654971 0.5116959 0.03467453
9  0.0029239766     18 0.3567251 0.5146199 0.03474917
10 0.0005847953     20 0.3508772 0.4970760 0.03429471
11 0.0001000000     25 0.3479532 0.5058480 0.03452394
```

(A) Row 11 (i.e., 25 splits) provides the lowest cross-validation error ("`xerror`").

(B) The best tree has `xerror` and standard error ("`xstd`") equal to `0.5058480 + 0.03452394 = 0.54037194`. Therefore, we would select the simplest tree that has an `xerror < 0.54037194`, which is row 3 (i.e., 3 splits). We call `prune()` and specify `cp` to be the same as the `cp` value in row 3. Ex.,

```
prune(tree, cp = 0.0233918129)
```

`prp()` - Plot `rpart`. Allows pruning with the `snip` argument.

- `x` - rpart object.
- `snip` - Makes an interactive plot for an `rpart` object. Click on nodes to delete them. When you quit, the object will automatically update with those nodes removed, provided you specify `y <- prp(x)$obj`.

`prune()` - Prune the tree; uses similar commands to the `control` argument of `rpart()`.

11.3.2. `randomForest` to model data using decision trees

While `rpart` generates a single decision tree, `randomForest` generates multiple trees at random, i.e., a forest. More specifically, a democratic forest; each tree will vote on an outcome and the majority will "win." As such, random forests provide one method of overcoming issues of overfitting in CART modeling.

Random forests are 'random' because trees are grown employing two techniques which ensure that the trees grow differently.

1) The first is *bagging*, which involves taking a random sample (with replacement) of the *observations* to be used to train the model.
    - This prevents the overall model from being overfitted to exceptional observations.
    - At the same time, *Out of bag* observations are used to test accuracy of your data.
2) The second is *subsetting*, which involves taking a random sample of the *variables* to be used to train the model; the available variables is re-sampled at each node. In general, only the square-root of total number of variables is sampled at each node.
    - This prevents a single variable from dominating the first node of every decision tree.

Limitations

- `randomForest` doesn't know how to deal with NA values. As such, it cannot be applied if there are may missing values, and you will have to exclude or assign values (either by prediction, or arbitrarily) for all observations.
    - `rpart` has the advantage that it knows how to employ surrogate variables.

Generating a model

`randomforest()`

- `x, formula` - Formula specifying the response variable and predictor. `x` is a dataframe (to be used with `y`). If the reponse variable is a factor, classification is assumed. Otherwise, regression is assumed.
- `y` - `y` is the response variable, if `x` is used instead of `formula`. If a factor, classification is assumed. Otherwise, regression is assumed.
- `data` - Optional data frame containing the variables in `formula`.
- `subset` - Optional index vector specifying which rows should be used.
- `na.action` - Function to be applied if NA values are found.
- `ntree` - Number of trees to compute. Should only be limited for computational reasons. Default = 500
- `importance` - Logical. Should the importance of each predictor be assessed? Generally, this should be TRUE.

Viewing results

`predict()` - Just like in base R or `rpart`, returns a vector of the predicted responses from a fitted `randomForest`

- `object` - `rpart` object. The model that you wish to extract the predictions from.
- `newdata` - Data frame. The "prediction/test" data frame, which contains all of the variables included in the original `rpart` model.
- `type` - String denoting the type of predicted value to be returned.
  - "`prob`" returns a matrix of probabilities of each outcome of the original outcome variable.
  - "`response`" (also accepts "`class`") returns predicted values, whether continuous or binary.
  - "`votes`" - the number of votes.

`varImpPlot()`

- `x` - `randomForest` object.
- `sort` - Logical. Should the variables be sorted by order of importance? Default = TRUE.
- `n.var` - Integer. How many variables should be stored? Ignored if sort = FALSE.

11.3.3. `party` to model data using conditional inference trees

party is another recursive partitioning package, but its main feature is ctree(), which constructs conditional inference trees, taking into account distributional properties of the variables. Conditional inference trees overcome the main limitations of the CART method, namely overfitting and bias toward factors with many levels. Instead of a "pure" approach, conditional inference trees use statistical tests to determine the importance of each variable.

NB: "*Ensembles of conditional inference trees have not yet been extensively tested, so this routine is meant for the expert user only and its current state is rather experimental. However, there are some things available in cforest that can't be done with randomForest, for example fitting forests to censored response variables (see Hothorn et al., 2006a) or to multivariate and ordered responses."* -- cforest help page.

Roughly, the `ctree` algorithm works as follows:

1) Test the global null hypothesis of independence between any of the input variables and the response.
   - Stop if this hypothesis cannot be rejected.
   - ELSE select the input variable with strongest association to the response. This association is measured by a p-value corresponding to a test for the partial null hypothesis of a single input variable and the response.
2) Implement a binary split in the selected input variable.
3) Recursively repeate steps 1) and 2).

Generating a model

`ctree()` - Make a single tree.

`formula` - formula object. Note that '`:`' and '`-`' will not work.

`data` - Optional data frame containing the variables in `formula`.

`subset` - Optional index vector specifying which rows should be used.

`weights` - Optional vector of weights to be used in the fitting process (*I don't know what this does*).

`controls` - `TreeControl` object. Can be achieved by feeding into the `ctree_control()` function. See below.

`scores` - Optional named list of scores to be attached to ordered factors. By default, the scores are `1:length(x)`. This may be changed to, for example: `list(x = c(1, 5, 6)`.

`ctree_control()`

`teststat` - Character specifying the test statistic to be applied.

`"quad"` `"max"`

`testtype` - Character specifying how to compute the distribution of the test statistic.

`"Bonferroni"` `"MonteCarlo"` `"Univariate"` `"Teststatistic"`

`mincriterion` - The cut-off alpha level that must be exceed to implement a split. Takes on the form of `1 - p`. Default = 0.95.

`minsplit` - Integer. Minimum number of observations that must exist in a node in order for a split to be attempted. Default = 20.

`minbucket` - Integer. Minimum number of observations that must exist in a terminal node. Default = 7.

`stump` - Logical. Should only stumps (a tree with only 3 nodes) be computed? Default = FALSE.

`nresample` - Integer. Number of Monte-Carlo replications to use when the distribution of the test statistic is simulated. Default = 9999

`maxsurrogate` - Integer. The number of surrogate splits to investigate. Note that only surrogate splits for ordered covariables have been implemented (version 1.1-2). Default = 0.

`mtry` - Integer. Number of input variables to be randomly sampled as candidates at each node for random forest-like algorithms. Default = 0 (indicating no random selection)

`savesplitstats` - Logical. Should two-sample statistics be saved for each node? Default = TRUE

`maxdepth` - Integer. Maximum depth of the tree. Default = 0 (indicating no limit)

`remove_weights` - Logical. Should weights attached to nodes be removed after fitting? Default = FALSE

`cforest()` - Make a forest. Inputs are identical to `ctree()` with the exception of `controls` and `newdata`.

`controls` - `TreeControl` object. Can be achieved by feeding into the `cforest_control()` function. Some notable parameters that should be considered:

- `mtry = 5` Change depending the number of predictor variables (usually $\sqrt{n}$)
- `ntree = 500` Increase if you have more variables.
- `mincriterion = 0.95` Regulates tree depth. To grow large trees, decrease this.

`cforest_control()` - Similar to `ctree_control()` with a few additional arguments and different defaults.

`teststat` - Character specifying the test statistic to be applied. Default = "`quad`"

`testtype` - Character specifying how to compute the distribution of the test statistic.

Default = "Teststatistic"

`mincriterion` - The cut-off alpha level that must be exceed to implement a split. Takes on the form of `1 - p`. Default = 0.95.

`mtry` - Integer. Number of input variables to be randomly sampled as candidates at each node for random forest-like algorithms. Default = 5.

`savesplitstats` - Logical. Should two-sample statistics be saved for each node? Default = TRUE.

`ntree` - Integer. Number of trees to grow. Default = 500.

`replace` - Logical. Should observations be sampled with or without replacement? Default = TRUE.

`fraction` - Numeric. What fraction of observations should be drawn if replace = FALSE? Default = 0.632.

`trace` - Logical. Should a progress bar be printing while the forest grows?

Viewing results

`predict()` - Uses `predict()` from base R.

`newdata` - Data frame to test the predictions. If NULL, will generate predictions on the original data.

11.4. Other Packages

11.4.1. `Amelia` to identify missing data

`missmap()`

`x` - Object.

11.4.2. `mice` for multiple imputation of missing values

For backgound on multiple imputation, see Error! Reference source not found.**.** Error! Reference source not found..

The basic function for imputing missing values is `mice()`. Other functions are included to assess the effect/appropriateness of the imputation.

11.4.2.1. Imputing data

`mice()` - Basic function for imputing missing values. Creates a new data frame containing `m` number of data sets.

`data` - The dataframe. Contains the missing entries and all variables that will be used to estimation and prediction.

`method` - Character vector. Method to use for imputatoin. "`pmm`", "`logreg`", "`polyreg`", "`cart`"

- `mice()` supports many other methods. See help file.

`m` - Numeric. Number of data sets to create. Default = 5

`maxit` - Numeric. Maximum number of iterations of imputation for each `m`. Default = 5

- Generally when using this function, you'll want to store it in a new object.

11.4.2.2. Visual diagnostics for imputations

```
pwplot(mice.data)

densityplot(mice.data)

xyplot(mice.data, x + y + z ~ .imp)
```

11.4.2.3. Creating and pooling models

* In general, you will want to create a model using your pooled MICE data and a model without imputation (i.e., list-wise deletion) so that you can compare them.

`with()` - Use the way you normally use the `with()` function on your MICE data frame; this allows you to create *m* number of models with a single `lm()` command and store them in a single object.

`lm()` - Model-fitting function (could be `glm()` or some other). Use the way you normally would. Combine with `with()` to create *m* number of models with a single command and store them in a single object.

`pool()` - Pool together the different models from a MICE model object into

11.4.2.4. Dealing with data that are not missing at random

The post argument of mice() allows you to adjust your argument.

11.4.3. `ordPens` to smooth ordinally scaled independent variables in regression

12. Sample Codes