

DATA MINING TUTORIAL

FUNCTIONS IN R

CONDITIONAL STATEMENTS

- ▶ Perform different commands in different situations
- ▶ *if (condition) command_if_true*
- Can add *else command_if_false* to end
- Group multiple commands together with braces {}
- *if (cond1) {cm1; cmd2} else if (cond2) {cmd3; cmd4}*

-
- ▶ Conditions use relational operators
 - ==, !=, <, >, <=, >=
 - Do not confuse = (assignment) with == (equality)
 - = is a command, == is a equation
 - ▶ Combine conditions with and (&&) and or (||)

LOOPS

- ▶ Most common type of loop is the *for* loop
- *for (x in v) {loop_commands;}*
- *v* is a vector, commands repeat for each value in *v*
- Variable *x* becomes each value in *v*, in order
- ▶ Example: adding the numbers 1-10
- *total = 0; for (x in 1:10) total = total+x;*

-
- ▶ Other type of loop is the *while* loop
 - *while (condition) {loop_commands;}*
 - Conditions identical to *if* statement
 - Commands are repeated until condition is false
 - Might execute commands 0 times if already false
 - ▶ *while* loops are useful when you don't know number of iterations

FOR LOOP PRACTICE

- ▶ `u1<- rnorm(30) # create a vector filled with random normal values`
- ▶ `print("This loop calculates the square of the first 10 elements of vector u1")`
- ▶ `usq<-0`
- ▶ `for(i in 1:10)`
- ▶ `{`
- ▶ `usq[i]<-u1[i]*u1[i] # i-th element of u1 squared into i-th position of usq`
- ▶ `print(usq[i])`
- ▶ `}`
- ▶ `print(i)`

NESTING FOR LOOPS

- ▶ *# nested for: multiplication table*
- ▶ *myamat = matrix(nrow=30, ncol=30) # create a 30 x 30 matrix (of 30 rows and 30 columns)*
- ▶ *for(i in 1:dim(myamat)[1]) # for each row*
- ▶ *{*
- ▶ *for(j in 1:dim(myamat)[2]) # for each column*
- ▶ *{*
- ▶ *myamat[i,j] = i*j # assign values based on position: product of two indexes*
- ▶ *}*
- ▶ *}*

WHILE LOOP PRACTICE

```
▶ readinteger <- function()  
▶ {  
▶   n <- readline(prompt="Please, enter your ANSWER: ")  
▶ }  
▶ response<-as.integer(readinteger())  
  
▶ while (response!=42)  
▶ {  
▶   print("Sorry, the answer to whatever the question MUST be 42");  
▶   response<-as.integer(readinteger());  
▶ }
```

WRITING YOUR OWN FUNCTIONS

- ▶ Writing functions in R is defined by an assignment like:
- ▶ `a<-function(arg1, arg2) {function_commands;}`
- ▶ Functions are R objects of type "function"
- ▶ Functions can be written in C/FORTRAN and called via `.C()` or `.Fortran()`

-
- ▶ Arguments may be have default values
 - Example: *my.pow<-function(base, pow=2) {return base^pow; }*
 - Arguments with default values become optional, should usually appear at end of argument list (though not required)

-
- ▶ Arguments are untyped
 - Allows multipurpose functions that depend on argument type
 - Use `class()`, `is.numeric()`, `is.matrix()`, etc. to determine type

DISTANCE FUNCTIONS IN R

- ▶ Euclidean distance of two vectors
- ▶ Example:
 - `x1<-rnorm(30)`
 - `x2<-rnorm(30)`
 - `dist(rbind(x1, x2))` # `rbind` forms a matrix of `x1` and `x2`
- ▶ Usage: This function computes and returns the distance matrix computed by using the specified distance measure to compute the distances between the rows of a data matrix.
 - `dist(x, method = "euclidean", diag = FALSE, upper = FALSE, p = 2)`

-
- `dist(x, method = "euclidean", diag = FALSE, upper = FALSE, p = 2)`
 - ▶ `x`: a numeric matrix, data frame or "dist" object.
 - ▶ `method`: the distance measure to be used. This must be one of "euclidean", "maximum", "manhattan", "canberra", "binary" or "minkowski". Any unambiguous substring can be given.
 - ▶ `diag`: logical value indicating whether the diagonal of the distance matrix should be printed by `print.dist`.
 - ▶ `upper`: logical value indicating whether the upper triangle of the distance matrix should be printed by `print.dist`.
 - ▶ `p`: the power of the Minkowski distance.

CALCULATING A DISTANCE MATRIX FOR GEOGRAPHIC POINTS USING R

- ▶ Here's an example of how to calculate a distance matrix for geographic points (expressed as decimal latitudes and longitudes) using R:
 - `df.cities <- data.frame(name = c("New York City", "Chicago", "Los Angeles", "Atlanta"), lat = c(40.75170, 41.87440, 34.05420, 33.75280), lon = c(-73.99420, -87.63940, -118.24100, -84.39360))`
 - `round(GeoDistanceInMetresMatrix(df.cities) / 1000)`

| | New York City | Chicago | Los Angeles | Atlanta |
|---------------|---------------|---------|-------------|---------|
| New York City | 0 | 1148 | 3945 | 1204 |
| Chicago | 1148 | 0 | 2808 | 945 |
| Los Angeles | 3945 | 2808 | 0 | 3116 |
| Atlanta | 1204 | 945 | 3116 | 0 |

-
- ▶ For example, the above distance matrix shows that the straight-line distance—accounting for curvature of the earth—between Los Angeles and NYC is 3,945 km.
 - ▶ And here's the code for the `GeoDistanceInMetresMatrix()` function that generates the matrix:

```
ReplaceLowerOrUpperTriangle <- function(m, triangle.to.replace){  
  if (nrow(m) != ncol(m)) stop("Supplied matrix must be square.")  
  if (tolower(triangle.to.replace) == "lower") tri <- lower.tri(m)  
  else if (tolower(triangle.to.replace) == "upper") tri <- upper.tri(m)  
  else stop("triangle.to.replace must be set to 'lower' or 'upper'.")  
  m[tri] <- t(m)[tri]  
  return(m)  
}
```

```

GeoDistanceInMetresMatrix <- function(df.geopoints){
  # Returns a matrix (M) of distances between geographic points.
  # M[i,j] = M[j,i] = Distance between (df.geopoints$lat[i], df.geopoints$lon[i]) and
  # (df.geopoints$lat[j], df.geopoints$lon[j]).
  # The row and column names are given by df.geopoints$name.

  GeoDistanceInMetres <- function(g1, g2){
    # Returns a vector of distances. (But if g1$index > g2$index, returns zero.)
    # The 1st value in the returned vector is the distance between g1[[1]] and g2[[1]].
    # The 2nd value in the returned vector is the distance between g1[[2]] and g2[[2]]. Etc.
    # Each g1[[x]] or g2[[x]] must be a list with named elements "index", "lat" and "lon".
    # E.g. g1 <- list(list("index"=1, "lat"=12.1, "lon"=10.1), list("index"=3, "lat"=12.1, "lon"=13.2))
    DistM <- function(g1, g2){
      require("Imap")
      return(iffelse(g1$index > g2$index, 0, gdist(lat.1=g1$lat, lon.1=g1$lon, lat.2=g2$lat, lon.2=g2$lon, units="m")))
    }
    return(mapply(DistM, g1, g2))
  }

  n.geopoints <- nrow(df.geopoints)

  # The index column is used to ensure we only do calculations for the upper triangle of points
  df.geopoints$index <- 1:n.geopoints

  # Create a list of lists
  list.geopoints <- by(df.geopoints[,c("index", "lat", "lon")], 1:n.geopoints, function(x){return(list(x))})

  # Get a matrix of distances (in metres)
  mat.distances <- ReplaceLowerOrUpperTriangle(outer(list.geopoints, list.geopoints, GeoDistanceInMetres), "lower")

  # Set the row and column names
  rownames(mat.distances) <- df.geopoints$name
  colnames(mat.distances) <- df.geopoints$name

  return(mat.distances)
}

```

K-MEANS EXAMPLE

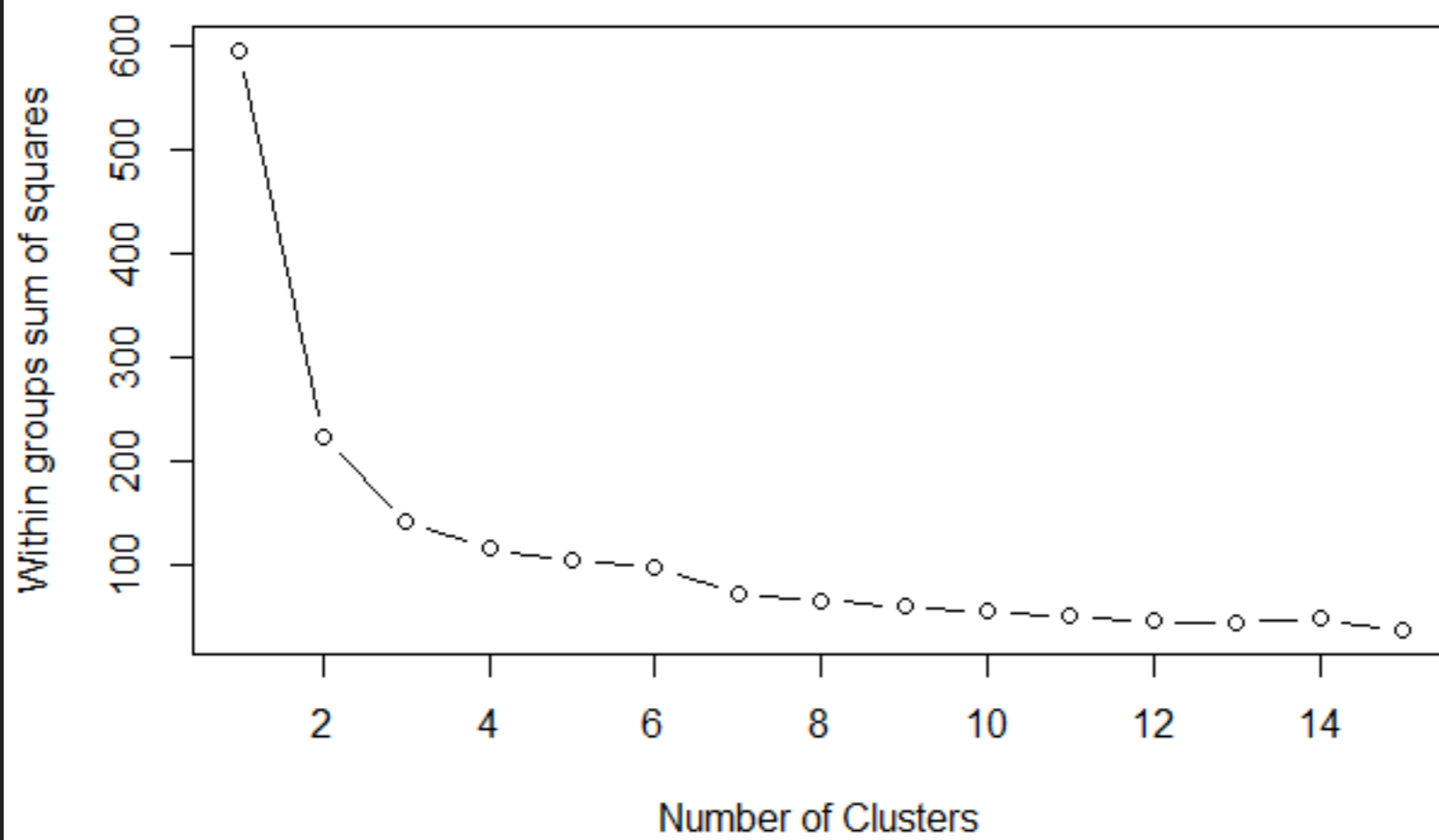
- ▶ <https://archive.ics.uci.edu/ml/datasets/Iris>
- ▶ Data Preparation
- Prior to clustering data, you may want to remove or estimate missing data and rescale variables for comparability
- `mydata <- na.omit(mydata) # listwise deletion of missing`
- `mydata <- scale(mydata[,1:4]) # standardize variables`

▶ Mydata

| | V1 | V2 | V3 | V4 |
|----|-------------|------------|-------------|-------------|
| 1 | -0.89767388 | 1.0286113 | -1.33679402 | -1.30859282 |
| 2 | -1.13920048 | -0.1245404 | -1.33679402 | -1.30859282 |
| 3 | -1.38072709 | 0.3367203 | -1.39346985 | -1.30859282 |
| 4 | -1.50149039 | 0.1060900 | -1.28011819 | -1.30859282 |
| 5 | -1.01843718 | 1.2592416 | -1.33679402 | -1.30859282 |
| 6 | -0.53538397 | 1.9511326 | -1.16676652 | -1.04652483 |
| 7 | -1.50149039 | 0.7979809 | -1.33679402 | -1.17755883 |
| 8 | -1.01843718 | 0.7979809 | -1.28011819 | -1.30859282 |
| 9 | -1.74301699 | -0.3551707 | -1.33679402 | -1.30859282 |
| 10 | -1.13920048 | 0.1060900 | -1.28011819 | -1.43962681 |
| 11 | -0.53538397 | 1.4898719 | -1.28011819 | -1.30859282 |
| 12 | -1.25996379 | 0.7979809 | -1.22344235 | -1.30859282 |
| 13 | -1.25996379 | -0.1245404 | -1.33679402 | -1.43962681 |
| 14 | -1.86378030 | -0.1245404 | -1.50682152 | -1.43962681 |
| 15 | -0.05233076 | 2.1817629 | -1.45014569 | -1.30859282 |
| 16 | -0.17309407 | 3.1042843 | -1.28011819 | -1.04652483 |
| 17 | -0.53538397 | 1.9511326 | -1.39346985 | -1.04652483 |
| 18 | -0.89767388 | 1.0286113 | -1.33679402 | -1.17755883 |
| 19 | -0.17309407 | 1.7205023 | -1.16676652 | -1.17755883 |

▶ Partitioning

- K-means clustering is the most popular partitioning method. It requires the analyst to specify the number of clusters to extract. A plot of the within groups sum of squares by number of clusters extracted can help determine the appropriate number of clusters. The analyst looks for a bend in the plot similar to a scree test in factor analysis



| | | | | | |
|-----|-------------|------------|-------------|-------------|---|
| 83 | -0.05233076 | -0.8164314 | 0.08010185 | 0.00174712 | 2 |
| 84 | 0.18919584 | -0.8164314 | 0.76021186 | 0.52588310 | 1 |
| 85 | -0.53538397 | -0.1245404 | 0.42015685 | 0.39484910 | 3 |
| 86 | 0.18919584 | 0.7979809 | 0.42015685 | 0.52588310 | 3 |
| 87 | 1.03453895 | 0.1060900 | 0.53350852 | 0.39484910 | 3 |
| 88 | 0.55148575 | -1.7389527 | 0.36348102 | 0.13278111 | 1 |
| 89 | -0.29385737 | -0.1245404 | 0.19345352 | 0.13278111 | 3 |
| 90 | -0.41462067 | -1.2776920 | 0.13677768 | 0.13278111 | 2 |
| 91 | -0.41462067 | -1.0470617 | 0.36348102 | 0.00174712 | 2 |
| 92 | 0.30995914 | -0.1245404 | 0.47683269 | 0.26381511 | 3 |
| 93 | -0.05233076 | -1.0470617 | 0.13677768 | 0.00174712 | 2 |
| 94 | -1.01843718 | -1.7389527 | -0.25995316 | -0.26032087 | 2 |
| 95 | -0.29385737 | -0.8164314 | 0.25012935 | 0.13278111 | 2 |
| 96 | -0.17309407 | -0.1245404 | 0.25012935 | 0.00174712 | 3 |
| 97 | -0.17309407 | -0.3551707 | 0.25012935 | 0.13278111 | 3 |
| 98 | 0.43072244 | -0.3551707 | 0.30680518 | 0.13278111 | 3 |
| 99 | -0.89767388 | -1.2776920 | -0.42998067 | -0.12928687 | 2 |
| 100 | -0.17309407 | -0.5858010 | 0.19345352 | 0.13278111 | 2 |
| 101 | 0.55148575 | 0.5673506 | 1.27029437 | 1.70518904 | 4 |
| 102 | -0.05233076 | -0.8164314 | 0.76021186 | 0.91898508 | 1 |

PURITY

- ▶ Computes the purity of a clustering given a known factor
- ▶ Within the context of cluster analysis, Purity is an external evaluation criterion of cluster quality. It is the percent of the total number of objects(data points) that were classified correctly, in the unit range $[0..1]$.

-
- ▶ To calculate Purity first create your confusion matrix This can be done by looping through each cluster c_i and counting how many objects were classified as each class t_i .
 - ▶ Then for each cluster c_i , select the maximum value from each row, sum them together and finally divide by the total number of data points.

| | | T1 | | T2 | | T3 |
|----|--|----|--|----|--|----|
| C1 | | 0 | | 53 | | 10 |
| C2 | | 0 | | 1 | | 60 |
| C3 | | 0 | | 16 | | 0 |

- Purity = $(53 + 60 + 16) / 140 = 0.92142$

BASIC UTILITY FUNCTIONS

- ▶ *length()* returns the number of elements
- ▶ *mean()* returns the sample mean
- ▶ *median()* returns the sample mean
- ▶ *range()* returns the largest and smallest values
- ▶ *unique()* removes duplicate elements
- ▶ *summary()* calculates descriptive statistics
- ▶ *diff()* takes difference between consecutive elements
- ▶ *rev()* reverses elements