CWRU DSCI351-451: Classification: Cluster Analysis

Prof.:Roger French, TA:Alan Curran

November. 16, 2017

Contents

12.2.0.1   Reading, Homeworks, Projects, SemProjects

- Homework:
    –
    –
- Readings:
    – ISLR 3 on Regression
    – ISLR 4 on Classification
- Projects: We will have four 2 week EDA projects
    – Project 3 Finished last week
- 451 SemProjects:
    – Report Outs 3 In Week 15a, 15b

12.2.0.2   Syllabus

12.2.0.3   GIS data for Wind on the globe

Windyty

Day:Date	Foundation	Practicum	Reading	Due
w1a:Tu:8/29/17	ODS Tool Chain	R, Rstudio, Git		
w1b:Th:8/31/17	Setup ODS Tool Chain	Bash, Git, Twitter	PRP4-33	HW1
W2a:Tu:9/5/17	What is Data Science	OIS:Intro2R	PRP35-64	**HW1 Due**
w2b:Th:9/7/17	Data Analytic Style, Git	Teatime:Intro2R, For loops	PRP65-93	HW2
w3a:Tu:9/12/17	Struct. of Data Analysis, SemProj	ISLR:Intro2R	PRP94-116	**HW2 Due**
w3b:Th:9/14/17	OIS3 Intro to Data	GapMinder, Dplyr, Magrittr	OI1-1.9,	
w4a:Tu:9/19/17	OIS3, Intro2Data part 2, Data	EDA: PET Degr.	EDA1-31	Proj1
w4b:Th:9/21/17	Hypothesis Testing	GGPlot2 Tutorial	EDA32-58	HW3
w5a:Tu:9/26/17	Distributions	SemProj RepOut1	R4DS1-3	**HW3 Due**
w5b:Th:9/28/17*	Wickham DSCI in Tidyverse	SemProj RepOut1	R4DS4-6	**SemProj1,**
w6a:Tu:10/3/17*	OIS Found. of Inference	Inference	R4DS7-8	**Proj1 Due**
w6b:Th:10/5/17	Summ. Stats & Vis.	Data Wrangling	R4DS9-16 Wrangle	
w7a:Tu:10/10/17		Midterm Review		
w7b:Th:10/12/17*	**MIDTERM EXAM**			HW4
w8a:Tu:10/17/17	Numerical Inference	Tidy Check Explore	OIS4	**HW4 Due**
w8b:Th:10/19/17	Algorithms, Models	Pairwise Corr. Plots	OIS5.1-4	Proj 2, HW5
Tu:10/24	**CWRU FALL BREAK**		R4DS17-21 Program	
w9b:Th:10/26/17	Categorical Infer	Predictive Analytics	OIS6.1,2	
w10a:Tu:10/31/17	SemProj	SemProj	OIS7	**SemProj2 HW5 Du**
w10b:Th:11/2/17	Lin. Regr.	Lin. Regr.	OIS8	**Proj.2 due**
w11a:Tu:11/7/17	Inf. for Regression	Curse of Dim.	OIS8	Proj 3
w11b:Th:11/9/17	Model Accuracy	Training Testing	ISLR3	HW6
w12a:Tu:11/14/17	Multiple Regr.	Mul. Regr. & Pred.	ISLR4	**HW6 due**
w12b:Th:11/16/17	Classification		ISLR4	
w13a:Tu:11/21/17	Classification	Clustering	ISLR6	**Proj 3 due**
Th:11/23/17	**THANKSGIVING**			Proj 4
w14a:Tu:11/28/17	Big Data	Hadoop		
w14b:Th:11/30/17	InfoSec	VerisDB		**SemProj3**
w15a:Tu:12/5/17*	SemProj ReportOut3			
w15b:Th:12/7/17*	SemProj ReportOut3			**Proj4**
	FINAL EXAM	**Tues.12/18, 12:00-3:00pm**	Olin 303	**SemProj4 due**

Figure 1: DSCI351/451 Syllabus

12.2.0.4 Nature: P values are just the tip of the iceberg

Statistics: P values are just the tip of the iceberg

P values, the 'gold standard' of statistical validity, are not as reliable as many scientists assume.

- Scientific method: Statistical errors

12.2.0.5 Data Analytics Pipeline

From Peng Leek

Data Analytics Pipeline

12.2.0.6 Cluster Analysis: An example of unsupervised learning

Unsupervised cluster analysis refers to algorithms

- that aim at producing homogeneous groups of cases from unlabeled data.

Unlabeled data:

- where we don't have a response Y variable,
- just a number of predictor variables.

The algorithm doesn't know beforehand

- what the membership to the groups is,
- and its goal is to find the structure of the data
- from similarities (or differences) between the cases;

A cluster is a group of cases, observations, individuals, or other units,

- that are similar to each other on the considered characteristics.
- These characteristics can be anything measurable or observable.

The choice of characteristics, or attributes, is important

- as different attributes will lead to different clusters.

We'll discuss the following:

- Distance measures
- Partition clustering with k-means,
    - including the steps in the computations of clusters, and
    - the selection of the best number of clusters
- Applications of k-means clustering

Clustering algorithms use distance measures between the cases

- in order to create these homogeneous groups of cases (we will discuss this next).

It is therefore important to

- transform the data on all dimensions to a similar scale
- before performing partition clustering with tools such as kmeans().

This is important because the distances are computed

- from all the dimensions we consider.
    - If one dimension has a range of values higher than the others
- (for example, values in centimeters compared to meters),
- differences in this dimension will be given much more importance compared to the others.

This is also true for agglomerative clustering,

- with tools such as hclust(),
- or any algorithm using distance measures,
    - such as nearest neighbor classification,
    - which we will also discuss.

12.2.0.7 Distance measures in clustering

12.2.0.7.1 Partitioning clustering algorithms

Partitioning clustering algorithms

- iteratively define k cluster centers and
- assign cluster membership
    - (or the probability of group membership)
- to cases based on distances between the case and the cluster.

12.2.0.7.2 Agglomerative clustering algoriths

Agglomerative clustering algorithms

- also create clusters based on distances,
- starting with each individual belonging to a separate cluster
- and the grouping clusters two by two.

The k-nearest neighbors algorithm also uses distance measures.

Consider only one attribute, for instance the height of individuals.

The distance of someone measuring 180 cm

- and someone measuring 170 cm will be 10
- on this sole dimension considering the algebraic difference
- between the two measures as our distance metric.

Things get a little more complicated

- when we add more attributes,
- such as weight (we will not consider variable scaling here).

Let's say the first individual is clearly overweight (90 kg),

- and the second has a normal weight (80 kg).

Considering only the sum of the difference

- between the measures as our distance metric,
- the difference between the individuals would be:
    - (180-170) + (90-100) = 0.

This clearly doesn't reflect the huge differences between these individuals;

- one is bigger and slimmer than the other.

Several distance metrics are available. Here are some examples:

12.2.0.7.3 Euclidean and Manhattan distances

The metrics closest to the sum of differences measure we just examined

- are the Euclidean and the Manhattan distances.
- The Manhattan distance

– sums the absolute value of the differences on all considered dimensions.
- The Euclidean distance
    – sums the squares of the differences and then performs a square root on the result.
- In case of only one dimension,
    – the result is equal to the difference between observations.

12.2.0.8 A quick fast example of k-means clustering

Use the iris dataset for Petal.Length and Petal.Width

12.2.0.8.1 What is k-Means Clustering?

k-means clustering

K Means Clustering is an unsupervised learning algorithm

- that tries to cluster data based on their similarity.

Unsupervised learning means that there is no outcome to be predicted,

- and the algorithm just tries to find patterns in the data.

In k means clustering,

- we have to specify the number of clusters
- we want the data to be grouped into.

The algorithm randomly assigns each observation to a cluster,

- and finds the centroid of each cluster.

Then, the algorithm iterates through two steps:

- Reassign data points to the cluster whose centroid is closest.
- Calculate new centroid of each cluster.

These two steps are repeated

- till the within cluster variation cannot be reduced any further.

The within cluster variation is calculated

- as the sum of the euclidean distance between the data points
- and their respective cluster centroids

```
library(datasets)
head(iris)
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1          5.1         3.5          1.4         0.2  setosa
## 2          4.9         3.0          1.4         0.2  setosa
## 3          4.7         3.2          1.3         0.2  setosa
## 4          4.6         3.1          1.5         0.2  setosa
## 5          5.0         3.6          1.4         0.2  setosa
## 6          5.4         3.9          1.7         0.4  setosa
```

Now library in ggplot2

```
library(ggplot2)
ggplot(iris, aes(Petal.Length, Petal.Width, color = Species)) + geom_point()
```
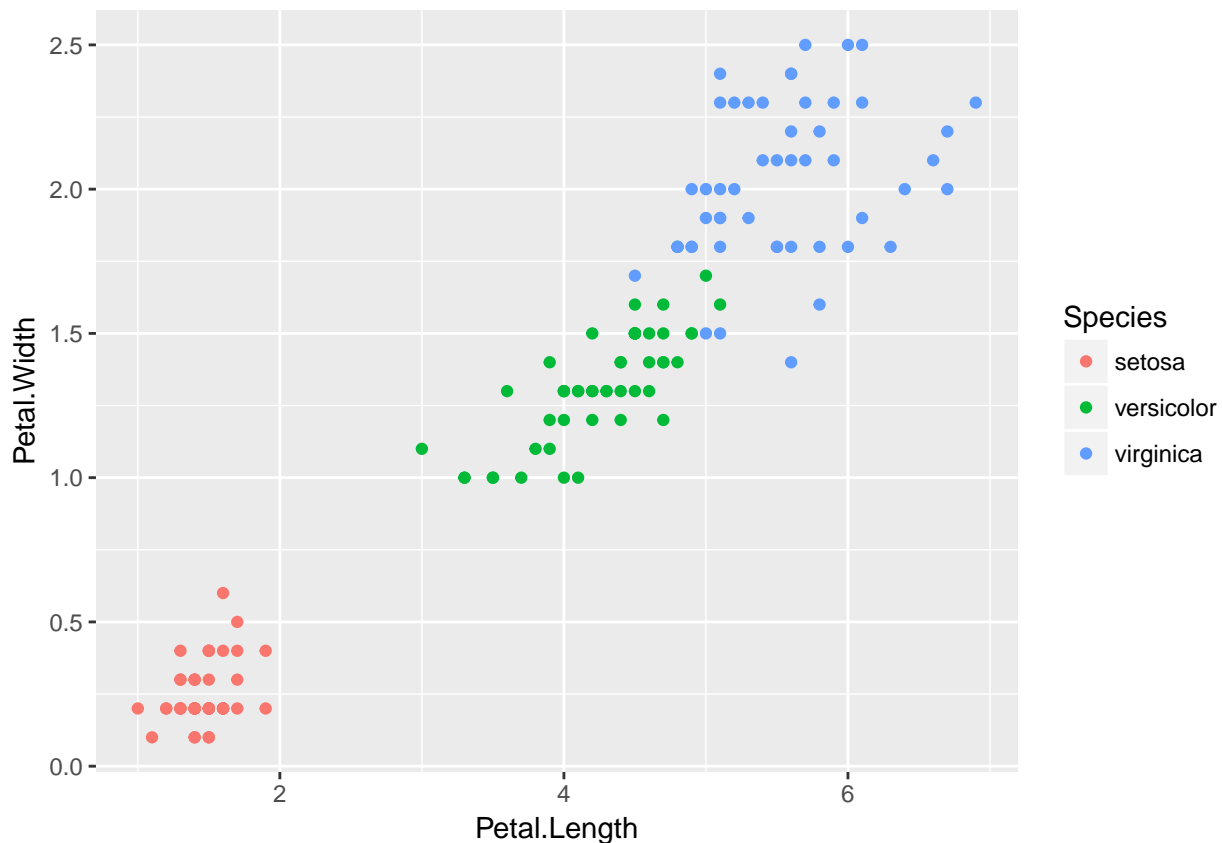
Now do the k-means clustering on these variables

```
set.seed(20)
irisCluster <- kmeans(iris[, 3:4], 3, nstart = 20)
irisCluster
```

```
## K-means clustering with 3 clusters of sizes 50, 52, 48
##
## Cluster means:
##   Petal.Length Petal.Width
## 1     1.462000    0.246000
## 2     4.269231    1.342308
## 3     5.595833    2.037500
##
## Clustering vector:
##   [1] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
##  [36] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
##  [71] 2 2 2 2 2 2 2 3 2 2 2 2 2 3 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 3 3 3 3 3
## [106] 3 2 3 3 3 3 3 3 3 3 3 3 3 3 3 2 3 3 3 3 3 3 2 3 3 3 3 3 3 3 3 3 3 3 2 3
## [141] 3 3 3 3 3 3 3 3 3 3
##
## Within cluster sum of squares by cluster:
## [1]  2.02200 13.05769 16.29167
##  (between_SS / total_SS =  94.3 %)
##
## Available components:
##
## [1] "cluster"      "centers"      "totss"        "withinss"
```

```
## [5] "tot.withinss" "betweenss"    "size"           "iter"
## [9] "ifault"
```

Since we know that there are 3 species involved,

- we ask the algorithm to group the data into 3 clusters,
- and since the starting assignments are random,
  - we specify nstart = 20.

This means that R will try 20 different random starting assignments and

- then select the one with the lowest within cluster variation.

We can see the cluster centroids,

- the clusters that each data point was assigned to,
- and the within cluster variation.

```r
table(irisCluster$cluster, iris$Species)
```

```
##
##     setosa versicolor virginica
## 1      50          0         0
## 2       0         48         4
## 3       0          2        46
```

As we can see, the data belonging to

- the setosa species got grouped into cluster 3,
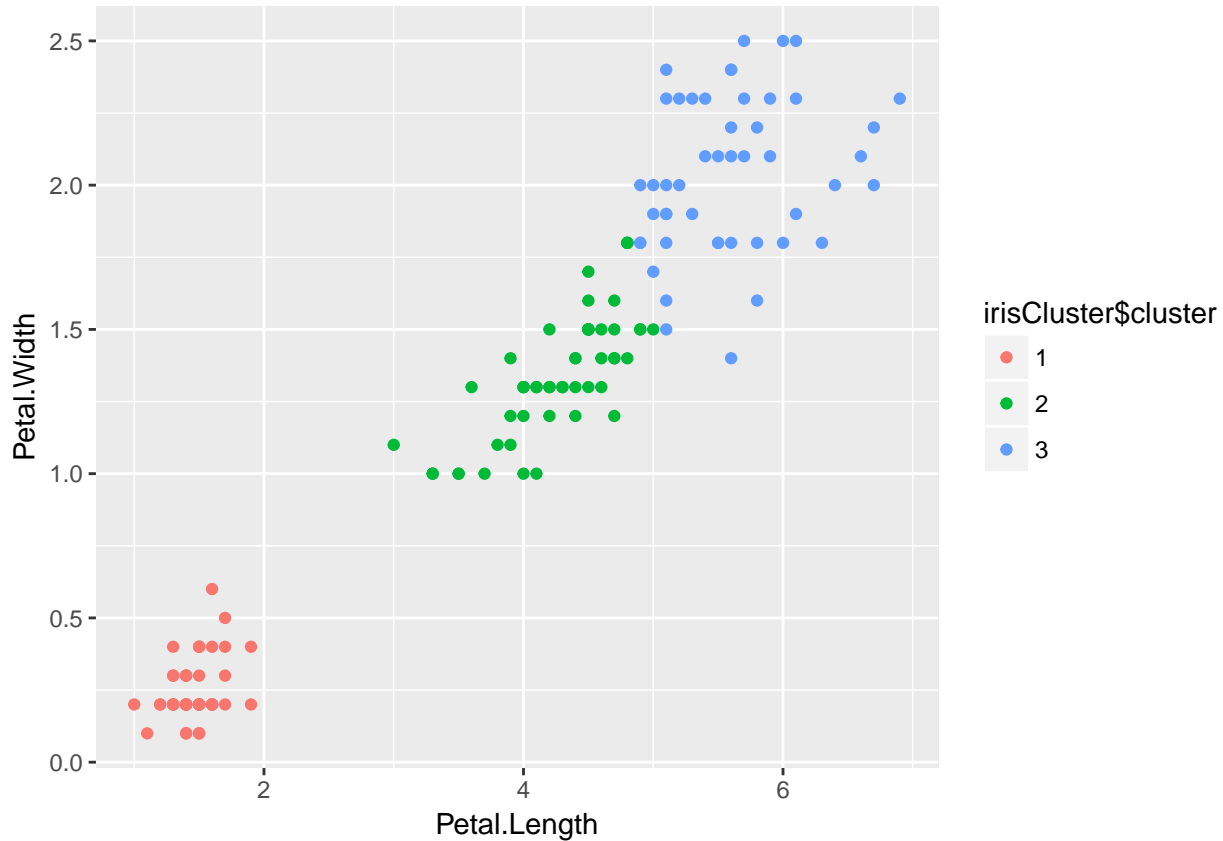- versicolor into cluster 2, and
- virginica into cluster 1.

The algorithm wrongly classified

- two data points belonging to versicolor
- and six data points belonging to virginica.

We can also plot the data to see the clusters:

```r
irisCluster$cluster <- as.factor(irisCluster$cluster)
ggplot(iris, aes(Petal.Length, Petal.Width, color = irisCluster$cluster)) + geom_point()
```

12.2.0.9 Partition clustering with k-means: A non-optimal "internal-workings" example

Perhaps the most widely used clustering family of algorithms is k-means.

In this section, we will examine how it works and

- ways to assess the quality of a clustering solution.

K-means is a partitioning algorithm

- that produces k (user-defined number) clusters of cases
- that are more similar to each other than to cases outside the cluster.

K-means starts by randomly initiating the centroid

- (the value of the considered dimensions) of each cluster.

From now, the process, aiming at creating homogenous clusters,

- is iterative until a final solution is found.
- For each case, the distance from the centroid of each cluster is computed,
    - and cases are assigned to the closest cluster.
- After this step, k-means computes the new values of the centroid of each cluster,
    - as the means of all the cases belonging to the cluster.
- The process stops when the distance between the cases
    - and the centroid is not decreasing anymore.

It is noteworthy that the final result at convergence

- depends upon the initial random values of the centroids.

For this reason, it is advocated to configure kmeans()

- to repeat the process several times
- and select the best solution (using the nstart argument).

The figure below illustrates this process.

We will only use the default algorithm with kmeans(),

- but the function provides several to choose from
- Hartigan-Wong, Lloyd, Forgy, and MacQueen
  - using the algorithm argument.

We will discuss this later.

The interested reader can inquire about the differences

- using the references provided in the ad-hoc R manual page (type ?kmeans in the console).
- *clustering process*

In what follows, we will build our own k-means implementation

- in order to betterunderstand its functioning.

There is a slight difference in our implementation

- in that we do not initiate the centroids randomly.

Instead we will attribute membership to the clusters randomly,

0 and then compute the centroids.

This is equivalent, but easier programmatically.

This implementation is intended only for pedagogical purposes.

It lacks several important features, and sometimes converges non-optimally.

For proper analyses,

- please use kmeans()
- or an implementation from another package available on CRAN.

12.2.0.9.1   Setting the centroids

First, we need a function that attributes membership to the clusters randomly.

We do this for all the observations at the same time,

- and return the result as vector clusters (see line 2).

Our function takes 2 arguments:

- the number of observations (numrows)
- and the number of clusters (k).

```r
set.random.clusters <- function(numrows, k) {
   clusters = sample(1:k, numrows, replace = T)
}
```

We then create a function that computes the centroids,

-bthat is, the means for each case on each dimension and each cluster.

Our function takes 2 arguments:

-bthe data frame on which to cluster the data (df), - and the current cluster assignments (clusters).

```
compute.centroids <- function(df, clusters) {
    means <- tapply(df[,1], clusters, mean)
    for (i in 2:ncol(df)) {
        mean.case <- tapply(df[,i], clusters, mean)
        means <- rbind(means, mean.case)
    }
    centroids <- data.frame(t(means))
    names(centroids) <- names(df)
    centroids
}
```

On line 2, we assign to vector means

- the mean values of attribute in column 1 for each of the clusters.

In a loop, we assign to vector mean.case

- the values of attribute in column i for each cluster,
- and append it to means (lines 4 and 5).

We then make a data frame

- from the transpose of object means and
- assign it to object centroids (line 8)

We name the columns of this object

- as the columns in the original data frame (line 9).

Finally we return centroids (line 9).

12.2.0.9.2 Computing distances to centroids

We then need a function that computes

- the distance between the actual values
- and the centroid of the clusters.

This function takes the data and the centroids as arguments.

It first creates a blank matrix that will contain the distances (line 2).

It then loops over the cases and the clusters

- (see the code blocks started on lines 3 and 4) and
- computes the squares of the differences
    - for the current case and current cluster,
- computes the square root of those, and
- assigns the value to the correct spot in the matrix (line 5).

It finally returns the matrix of Euclidean distances (line 8).

```
euclid.sqrd <- function(df, centroids) {
    distances <- matrix(nrow = nrow(df), ncol = nrow(centroids))
    for (i in 1:nrow(df)) {
        for (j in 1:nrow(centroids)) {
            distances[i,j] <- sum((df[i,] - centroids[j,])^2)
        }
    }
    distances
}
```

12.2.0.9.3 Computing the closest cluster for each case

We now need a function that will compare, for each case,

- its distance to each of the clusters and
- select the cluster where this value is minimum
- that is, assign the case to a cluster based on this comparison.

For this purpose, on line 2,

- we simply use the which.min() function,
  - which indicates in which column the minimal value is,
- as an argument of the apply() function,
  - which applies a function to each row of its input.

We wrap this in the cbind() function,

- which allows it to return column-shaped vectors.

If the number of returned clusters is less than expected

- (tested on line 3),
- we restart the process by assigning cases to random clusters,
- using the set.random.clusters() function (line 5).

In other words, we take the precaution of

- restarting the process of setting the centroids randomly
- if we find an empty cluster.

```
assign <- function(distances) {
   clusters <- data.frame(cbind(c(apply(distances, 1, which.min))))
   if (nrow(unique(clusters)) < ncol(distances)) {
   #precaution in case of empty cluster
   clusters <- set.random.clusters(nrow(distances),ncol(distances))
   }
   clusters
}
```

12.2.0.9.4 Tasks performed by the main function

We now almost have everything we need for our basic k-means implementation.

We finally need to wrap this together in a main function,

- which we call kay.means() .

Here we set initial cluster value (line 2)

- and then iterate over the computation of centroids (line 7),
- the calculation of distances (line 8),
- and the re-assignment of clusters (line 11).

Notice the code block is contained in a while loop,

- which stops when the sum of squares of the distances
  - (that is the total sum of squares within clusters)
- is the same twice in a row
  - (when ss.old equal to ss this value is set on line 10), and
- output the clustering solution (line 14).

```
kay.means <- function(df, k) {
    clusters = set.random.clusters(nrow(df),k)
    ss.old = 1e100
    ss = 1e99
    while (ss != ss.old) {

        centroids <- compute.centroids(df, clusters)
        distances <- euclid.sqrd(df, centroids)
        ss.old <- ss
        ss <- sum(distances)
        clusters <- assign(distances)
    }
    names(clusters) <- "Clusters"
    clusters
}
```

Let's try this using a very popular dataset,

- which we have already encountered in the previous chapter:
- the iris dataset, where observations are 150 iris flowers.

Attributes are the species of the flowers and their petal and sepal length and width.

So, in this case, we know the groups beforehand

- and are interested in knowing whether
- k-means can predict it from the other attributes.

Let's start by having a look at the correct classification,

- which is in the 5th column of the data set.

You will see in your console that

- cases 1 to 50 are of species Setosa,
- cases 51 to 100 are of species Versicolor, and
- cases 101 to 150 of species Virginica.

We will use our knowledge of the dataset

- to determine the number of clusters.

We select three clusters as we know there are three species.

We will talk later about determining the number of clusters

- when this information is not available.

```
iris[5]
```

```
##        Species
## 1       setosa
## 2       setosa
## 3       setosa
## 4       setosa
## 5       setosa
## 6       setosa
## 7       setosa
## 8       setosa
## 9       setosa
## 10      setosa
```

```
## 11     setosa
## 12     setosa
## 13     setosa
## 14     setosa
## 15     setosa
## 16     setosa
## 17     setosa
## 18     setosa
## 19     setosa
## 20     setosa
## 21     setosa
## 22     setosa
## 23     setosa
## 24     setosa
## 25     setosa
## 26     setosa
## 27     setosa
## 28     setosa
## 29     setosa
## 30     setosa
## 31     setosa
## 32     setosa
## 33     setosa
## 34     setosa
## 35     setosa
## 36     setosa
## 37     setosa
## 38     setosa
## 39     setosa
## 40     setosa
## 41     setosa
## 42     setosa
## 43     setosa
## 44     setosa
## 45     setosa
## 46     setosa
## 47     setosa
## 48     setosa
## 49     setosa
## 50     setosa
## 51 versicolor
## 52 versicolor
## 53 versicolor
## 54 versicolor
## 55 versicolor
## 56 versicolor
## 57 versicolor
## 58 versicolor
## 59 versicolor
## 60 versicolor
## 61 versicolor
## 62 versicolor
## 63 versicolor
## 64 versicolor
```

```
## 65  versicolor
## 66  versicolor
## 67  versicolor
## 68  versicolor
## 69  versicolor
## 70  versicolor
## 71  versicolor
## 72  versicolor
## 73  versicolor
## 74  versicolor
## 75  versicolor
## 76  versicolor
## 77  versicolor
## 78  versicolor
## 79  versicolor
## 80  versicolor
## 81  versicolor
## 82  versicolor
## 83  versicolor
## 84  versicolor
## 85  versicolor
## 86  versicolor
## 87  versicolor
## 88  versicolor
## 89  versicolor
## 90  versicolor
## 91  versicolor
## 92  versicolor
## 93  versicolor
## 94  versicolor
## 95  versicolor
## 96  versicolor
## 97  versicolor
## 98  versicolor
## 99  versicolor
## 100 versicolor
## 101  virginica
## 102  virginica
## 103  virginica
## 104  virginica
## 105  virginica
## 106  virginica
## 107  virginica
## 108  virginica
## 109  virginica
## 110  virginica
## 111  virginica
## 112  virginica
## 113  virginica
## 114  virginica
## 115  virginica
## 116  virginica
## 117  virginica
## 118  virginica
```

```
## 119  virginica
## 120  virginica
## 121  virginica
## 122  virginica
## 123  virginica
## 124  virginica
## 125  virginica
## 126  virginica
## 127  virginica
## 128  virginica
## 129  virginica
## 130  virginica
## 131  virginica
## 132  virginica
## 133  virginica
## 134  virginica
## 135  virginica
## 136  virginica
## 137  virginica
## 138  virginica
## 139  virginica
## 140  virginica
## 141  virginica
## 142  virginica
## 143  virginica
## 144  virginica
## 145  virginica
## 146  virginica
## 147  virginica
## 148  virginica
## 149  virginica
## 150  virginica
```

12.2.0.10   Validation of our clustering method

Our goal now is to assess the quality of our clustering solution.

We are lucky, as we have the right answer already (which is not always the case).

In order to check how well our clustering solution did,

- we first append a column to the iris dataset
  - with the clustering solution of our implementation of k-means, and
  - observe the convergence between the clustering and the species
  - by creating a cross-tabulation of the data.

```
set.seed(1)
irisClust <- cbind(iris, kay.means(iris[1:4], 3))
tableClust <- table(unlist(irisClust[5]), unlist(irisClust[6]))
tableClust
```

```
##
##               1  2  3
##   setosa      0  0 50
##   versicolor  3 47  0
##   virginica  36 14  0
```

Our implementation of k-means did a good job on this dataset:

- flowers of the Setosa species are classified in cluster 3,
- flowers of the Versicolor species are **almost all** classified in cluster 2,
- while flowers of the Virginica species **are mostly** classified in cluster 1,
  - with a larger degree of misclassifications in cluster 2 (about a third).

12.2.0.10.1 Other measures of correctness: Cohen's Kappa

As we have information about the correct solution,

- we can compute indices of the correctness of our cluster solution.

For instance, we can compute Cohen's kappa,

- which assesses the agreement of our clustering solution with the correct one.

See the paper A coefficient of agreement for nominal scales, by Cohen (1960) for a description of the measure.

The closer its value is to 1, the better the agreement.

This index is in the psych package,

- which needs installing and loading
- before we proceed with the analysis.

```
# if(!require("psych")) install.packages("psych")
library(psych)
```

```
##
## Attaching package: 'psych'

## The following objects are masked from 'package:ggplot2':
##
##     %+%, alpha
```

The index requires that categories are given the same name

- in both the classification and the original solution.

We therefore need to recode our data to proceed:

```
irisClust <- cbind(irisClust, rep(0,nrow(irisClust)))
names(irisClust[7]) <- "Species.recode"
irisClust[7][irisClust[5] == "setosa"] = 3
irisClust[7][irisClust[5] == "versicolor"] = 2
irisClust[7][irisClust[5] == "virginica"] = 1
```

We now have the data of both

- our clustering solution (in column 6)
- and the correct solution (in column 7) in the same format.

We can now apply Cohen's kappa to our data:

```
kappa=cohen.kappa(cbind(irisClust[6],irisClust[7]))
```

```
## Warning in any(abs(bounds)): coercing argument of type 'double' to logical
```

Cohen Kappa and Weighted Kappa correlation coefficient

- and confidence boundary output is as follows:

12.2.0.10.2 rand index,

We can also compute the rand index,

- which is another measure of agreement.

We can find it in the flexclust package.

- Please install and load it before you proceed:

Here we can simply reuse the cross-tabulation we produced earlier:

```r
# if(!require("flexclust")) install.packages("flexclust")
library(flexclust)
```

```
## Loading required package: grid
```

```
## Loading required package: lattice
```

```
## Loading required package: modeltools
```

```
## Loading required package: stats4
```

```r
randIndex(tableClust)
```

```
##       ARI
## 0.7163421
```

The value of the index is .716,

- indicating also a good agreement.

Try using the kay.means() function with a different seed,

- such as 3 for instance,
- and notice the differences.

12.2.0.10.3 Summary

While clustering solutions using k-means

- are always dependent on initial centroid values,
- our implementation is even more vulnerable.

It is always a good idea to perform a cluster analysis

- with different seeds,
- which we will do later using the nstart argument with kmeans().

Nevertheless, our custom function fulfilled its purpose,

- which is to give the reader a sense of how k-means works.

12.2.0.11 Cites

- K-means clustering in R
- Learning Predictive Analytics with R, Eric Mayor, Packtpub 2015