**ANLY 506**

**Preparing for Quiz 3: Cluster analysis, Transformations, PCA and Dimension Reduction**

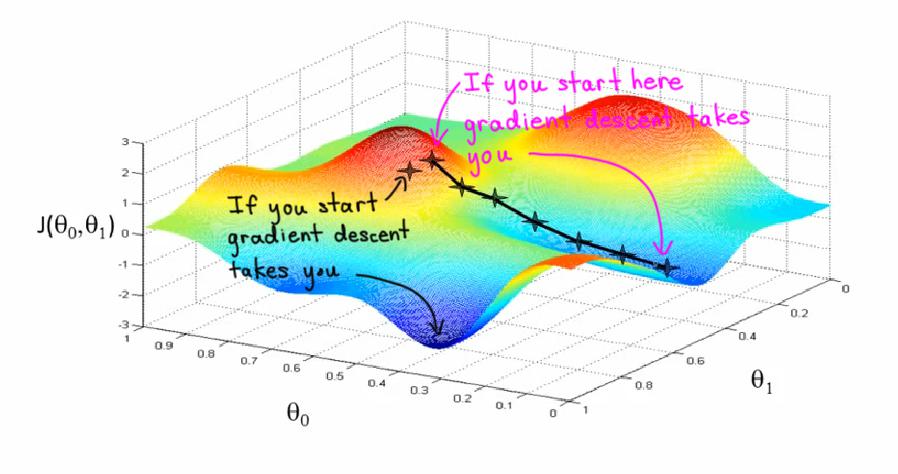
**Principle Component Analysis and Dimension Reduction**

Be sure that you’ve thoroughly reviewed Lectures 11 and 14 including the R commands/scripting. R-Bloggers also has a number of webpages on PCA, e.g. <https://www.r-bloggers.com/principal-component-analysis-using-r/>that you can review. Keep in mind that many and perhaps all websites you visit will have a slightly different way of conducting PCA and/or slightly different syntax to fit the many R commands that are available for you.

**Transformations**

It is common to do transformations, particularly normalization, as part of pre-processing data. Pre-processing data often falls in the realm of exploratory data analysis. For example, there are several types of analyses that either greatly benefit or are totally enabled (meaning that without this they cannot be completed) by performing data transformations. One of these types of analysis is multivariate linear regression. Don’t confuse multivariate linear regression with multiple linear regression, which may be more familiar. Without going into the details of multivariate linear regression let’s just say that if the range of the variables varies too greatly then the overall process may fail to find an optimum solution.

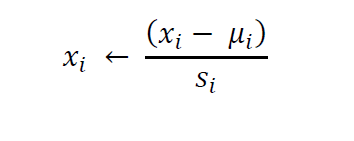
This is illustrated in the graphic below where the solution you obtain depends on where you start. This is an illustration of a cost function with multiple parameters. An example would be trying to predict the price of a house given variables such as number of rooms, number of floors, age, size, lot size, and so on. This picture is from Dr. Ng at Stanford University.



Obviously, all of the parameters in this example, predicting the price of a house, have very different units of measure and scales. So, in order not to start with the sort of lumpy looking model shown in the image above we’ll transform our variables. For multivariate linear regression this is referred to as “feature scaling”.

There are several ways to do this. We’ll look at some common ways. Keep in mind that the goal is to get all variables on a common scale, typically from 0 to 1. -1 to 1, or close to that and a relatively smooth surface contour, i.e. not lumpy. To conduct feature scaling let’s look at the size of houses and the number of bedrooms each house has. Size cannot be less than 0 but could run into the 1,000’s of square feet. Likewise, the number of bedrooms cannot be 0 but certainly will not extend into the 1,000’s, or at least not for a house. One way to scale these would be to simply take their inverse, e.g. 1/size or 1/bedrooms. This will yield values on a scale from 0 to 1. However, that only scales the values and will not reduce the “lumpiness” of the surface.

If we knew that none of the houses in our data set exceeded a size of 2000, we could just divide all sizes by 2000 to get all values for size on a scale from 0 to 1. But size doesn’t really have a maximum. Another thing we can do is to calculate the average size of the values for the size variable and subtract that from each value for size in our data set. That still doesn’t get us to a scale from -1 to 1. So, there are two different ways to approach this. One way is to find the range of values, maximum minus minimum, and divide the difference between the size and the average size by the range. An alternative way is to find the standard deviation for the variable of interest, then divide by that. The equation is:

****

Where the arrow simply means that the value of “is assigned” the value given by the equation, is the mean, and is the range or standard deviation. You’ll know whether to use the range or the standard deviation depending of the variable and the nature of the analysis you are conducting.

How much difference does it make which way we choose to conduct feature scaling? Let’s check out a data set. You can download the dataset used for this example from <https://wiki.csc.calpoly.edu/datasets/wiki/Houses>Just get the .csv file in the section “Data”. It is really simple to determine that the average number of bedrooms is 3, assuming you round down rather than up; and, the average size is 1,755 square feet. The standard deviations are 0.86 bedrooms and 819 square feet. However, the range is a bit odd. The minimum number of bedrooms listed is 0, which doesn’t make sense. The maximum number is 10. Similarly, the minimum size is 120 square feet, which is really small for a house. The maximum size is 6,800 square feet. So the ranges are 10 bedrooms and 6,680 square feet.

If we use the range to complete our feature scaling, e.g. for number of bedrooms, the max value is 0.686 and the min -0.314. The difference is 1. If we use the standard deviation to complete our feature scaling for number of bedrooms the max value is 8.014 and the min value is -3.672. So, we’ve scaled the variable either way but obviously the method that is best for this variable and this data set is to use the range. That method yields the desired scale from -1 to 1. A rule of thumb is that scaled values probably should be in the range from -3 to +3. So, in this case, scaling using the standard deviation does not help us. I invite you to do the same check on methods for the size variable.

You could also say that it is easy to scale any single value in the data set. But we have 781 observations. So, feature scaling by hand would be tedious at best. And, depending on the program used you might need to add columns that are side-by-side, e.g. in Excel. So, my suggestion is to just use R again.

Here is another example of why scaling is an important part of transformations. Consider the article “Practical Guide to Principal Component Analysis (PCA) in R & Python at: [https://www.analyticsvidhya.com/blog/2016/03/practical-guide-principal-component-analysis-python/.](https://www.analyticsvidhya.com/blog/2016/03/practical-guide-principal-component-analysis-python/) In this article there is a section titled “Why is normalization of variables necessary?” This section includes a plot of a principle component analysis using unscaled and scaled variables. The answer is that the computer doesn’t have any idea what the variables are. It is only performing mathematical computations on values. So, if one set of values is on much greater scale, or covers a much greater range, then the computer sees it as having a much greater dispersion. But in reality this may not be true. Only by properly transforming/scaling

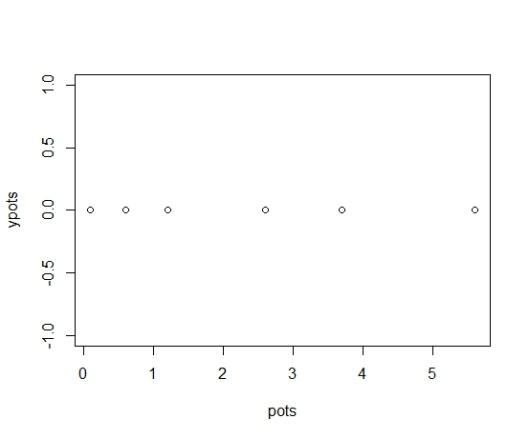
variables are we able to determine which components are truly principle components. We’ll do more with this in our work on cluster analysis next.

**Cluster Analysis Exercise**

Before we start there are a number of excellent references for cluster analysis, e.g. Chapter 8 from Tan, et al’s book on data mining can be found at: [http://www-users.cs.umn.edu/~kumar/dmbook/ch8.pdf.](http://www-users.cs.umn.edu/~kumar/dmbook/ch8.pdf) When we perform K-means clustering by hand we start by initializing some points that we think are going to be close to the centroids of the clusters. For example if we had the points:

{1.2, 5.6, 3.7, 0.6, 0.1, 2.6}

which are shown in the plot below.



We might start by picking the points 2 and 5 as the initial centroids. (Note that this is a worked example from our friends at Penn State from their STAT 897D course.)

The question is which points are closer to 2 and which are closer to 5. You have had the lecture and could calculate this for yourself. I’ll just tell you that {1.2, 0.6, 0.1, 2.6} are closer to 2 and {5.6, 3.7} are closer to 5. Once we have the initial clusters then we can refine the location of the centroids. I’m sure you could also do the calculations to refine the location of the centroids too. I’ll just tell you that they are 1.125 and 4.65. If you pick different locations for the initial centroids, e.g. 0.8 and 3.8, you’ll get a different answer. Unfortunately, neither of these yield the optimal locations of the centroids for these points. This is an important point to keep in mind.

Let's look at a more complicated example using R. Let's go step-by-step through an example of K-means clustering using diabetes data from UC Irvine's data repository. Please download the file "DiabetesData.csv" from Moodle.

First we'll read in the data and look at some summary statistics and the structure of the data set. Don't forget to put your data in your working directory or change your directory in R to where you have saved your data.

> diabetes=read.csv("DiabetesData.csv")  
> summary(diabetes)

Number.of.times.pregnant Plasma.glucose.concentration Blood.pressure   
 Min. : 0.000 Min. : 0.0 Min. : 0.00   
 1st Qu.: 1.000 1st Qu.: 99.0 1st Qu.: 62.00   
 Median : 3.000 Median :117.0 Median : 72.00   
 Mean : 3.845 Mean :120.9 Mean : 69.11   
 3rd Qu.: 6.000 3rd Qu.:140.2 3rd Qu.: 80.00   
 Max. :17.000 Max. :199.0 Max. :122.00   
 Skin.fold.thickness Insulin.Level BMI   
 Min. : 0.00 Min. : 0.0 Min. : 0.00   
 1st Qu.: 0.00 1st Qu.: 0.0 1st Qu.:27.30   
 Median :23.00 Median : 30.5 Median :32.00   
 Mean :20.54 Mean : 79.8 Mean :31.99   
 3rd Qu.:32.00 3rd Qu.:127.2 3rd Qu.:36.60   
 Max. :99.00 Max. :846.0 Max. :67.10   
 Diabetes.pedigree.function Age Class   
 Min. :0.0780 Min. :21.00 Min. :0.000   
 1st Qu.:0.2437 1st Qu.:24.00 1st Qu.:0.000   
 Median :0.3725 Median :29.00 Median :0.000   
 Mean :0.4719 Mean :33.24 Mean :0.349   
 3rd Qu.:0.6262 3rd Qu.:41.00 3rd Qu.:1.000   
 Max. :2.4200 Max. :81.00 Max. :1.000

The function glimpse from the package dplyr is an alternative to the str function in base R. Let's run glimpse to get an idea of the structure of the data. If you haven't downloaded the dplyr package to your computer, do this using the install.packages() command.

> install.packages('dplyr')

> library(dplyr)

> glimpse(diabetes)

Observations: 768  
Variables: 9  
$ Number.of.times.pregnant <int> 6, 1, 8, 1, 0, 5, 3, 10, 2, 8, 4,...  
$ Plasma.glucose.concentration <int> 148, 85, 183, 89, 137, 116, 78, 1...  
$ Blood.pressure <int> 72, 66, 64, 66, 40, 74, 50, 0, 70...  
$ Skin.fold.thickness <int> 35, 29, 0, 23, 35, 0, 32, 0, 45, ...  
$ Insulin.Level <int> 0, 0, 0, 94, 168, 0, 88, 0, 543, ...  
$ BMI <dbl> 33.6, 26.6, 23.3, 28.1, 43.1, 25....  
$ Diabetes.pedigree.function <dbl> 0.627, 0.351, 0.672, 0.167, 2.288...  
$ Age <int> 50, 31, 32, 21, 33, 30, 26, 29, 5...  
$ Class <int> 1, 0, 1, 0, 1, 0, 1, 0, 1, 1, 0, ...

So, we have a data set with 768 observations and 9 variables. Before we do too much we'll want to perform the appropriate transformations on the data. For example, we may want to scale our data appropriate. We perform this step to get all variables in a similar range. This is easy in R using the scale() command. The scale() command in R, by default, uses the scaling function described earlier 𝑥𝑖 ← (𝑥𝑖− 𝜇𝑖)/𝑠𝑖, where is 𝜇𝑖 the mean and 𝑠𝑖 is the standard deviation.

For this quiz to scale and standardize our data we'll simply use the command:

> df=scale(diabetes[, -9])

In this case we use [ , -9] to not include the last column which is our Class variable. This variable is either 0 or 1 and indicates the prescence of diabetes. If we transform this variable, we will not know which patients had diabetes in our dataset. This variable also already falls within an acceptable range. Note that we've assigned our scaled, standardized data to the object df. So, looking at the structure of df we see:

> glimpse(df)

num [1:768, 1:8] 0.64 -0.844 1.233 -0.844 -1.141 ...  
 - attr(\*, "dimnames")=List of 2  
 ..$ : NULL  
 ..$ : chr [1:8] "Number.of.times.pregnant" "Plasma.glucose.concentration" "Blood.pressure" "Skin.fold.thickness" ...  
 - attr(\*, "scaled:center")= Named num [1:8] 3.85 120.89 69.11 20.54 79.8 ...  
 ..- attr(\*, "names")= chr [1:8] "Number.of.times.pregnant" "Plasma.glucose.concentration" "Blood.pressure" "Skin.fold.thickness" ...  
 - attr(\*, "scaled:scale")= Named num [1:8] 3.37 31.97 19.36 15.95 115.24 ...  
 ..- attr(\*, "names")= chr [1:8] "Number.of.times.pregnant" "Plasma.glucose.concentration" "Blood.pressure" "Skin.fold.thickness" ...

That is, we are left with a 768 x 8 array of scaled data. Now we can begin to consider how many clusters we should have. There are several ways to look at this:

1. Start with a large number of clusters, k, and keep reducing k until it no longer affects the description length (see ["MDL principle for robust vector quantisation"](http://link.springer.com/article/10.1007/s100440050015) by Horst Bischof, Ales Leonardis, and Alexander Selb in *Pattern Analysis and Applications* vol. 2, p. 59-72, 1999. for example).
2. Start with one cluster then keep splitting clusters until some predefined criteria is reached, e.g. the points in each cluster have a Gaussian distribution (see ["Learning](http://books.nips.cc/papers/files/nips16/NIPS2003_AA36.pdf) [the *k* in *k*-means"](http://books.nips.cc/papers/files/nips16/NIPS2003_AA36.pdf) by Greg Hamerly and Charles Elkan (NIPS 2003)).

What we’ll do is let R do a lot of the work for us. We’ll look at the number of clusters two ways. First we’ll use a small R script as shown below to plot how the variation in cluster centers is reduced with increasing number of clusters using the sum of squares criteria. Then, we’ll invoke the NbClust package in R. The reference for you to check on this script and the overall process is at: <https://www.r-bloggers.com/k-means-clustering-from-r-in-action/>

So, first the small R script from Tal Galili at R-Bloggers:

wssplot <- function(data, nc=15, seed=1234){

wss <- (nrow(data)-1)\*sum(apply(data,2,var))

for (i in 2:nc){

set.seed(seed)

wss[i] <- sum(kmeans(data, centers=i)$withinss)}

plot(1:nc, wss, type="b", xlab="Number of Clusters",

ylab="Within groups sum of squares")

return(wss)}

The easiest way to implement this is to start a new R script in RStudio. If you highlight the entire script and click on the “Source” icon it will source the entire script so you can use it. Alternatively, you can copy the script above into a blank R file, save the file in your working directory as wssplot.R, navigate back into your main file and run the command

> source('wssplot.R')

Note that either way, you have to source this script before you can use it.

If you have never executed a script before you should take some time to check out how to do this. Consider this script and you should see that to execute it you will need to include a parameter for the data. That is, in the function() command data is set-up to use “data” as a parameter for your input. You can use this script for any data set by entering the command:

> nc1 <- wssplot(name of your data object)

In this case the name of the data object after we scaled and standardized the data is “df”. And, we want to assign the output of our script to an object, e.g. nc1 for the first output number of clusters. So the command to execute your wssplot function script for this example is:

> nc1 <- wssplot(df)

Plots should appear in your “Plots” window in RStudio. Note that you set the maximum number of clusters to be 15 in the function definition, i.e. nc=15. So, each plot will have a maximum number of clusters equal to 15. Now let’s see how this compares to the NbClust package output.

If you haven’t already installed the NbClust package you must do that first using

> install.packages('NbClust')

Then, to attach it, enter

> library(NbClust)

We’ll use our already scaled, standardized data assigned to the object df again. We’ll ouput the answer from the NbClust() command to another object “nc2” which just stands for the second output “number of clusters.” The parameters we’ll set in our NbClust() command include the minimum number of clusters, min.nc=2, the maximum number of clusters equal to 15 again, max.nc=15, and the method we’ll use is k-means, i.e. method=”kmeans”. The command looks like this:

> nc2 <- NbClust(df, min.nc=2, max.nc=15, method="kmeans")

Note that it may take a few minutes for R to complete the required calculations for this command. Also, this generates side-by-side plots. So, to complete this exercise you’ll need to execute the par() command to go back to a single plot mode, i.e.

> par(mfrow=c(1,1))

You’ll get the output from this command and the majority rule “best” number of clusters. But you’ll also get a lot more information. Let’s take a minute to look at some of the information generated by NbClust(). In addition to the majority rule determination, NbClust() looks at the best number of clusters using a wide variety of methods. To list this information use the command:

> nc2$Best.nc

This means that there are a lot of ways to consider what the best number of clusters is. Looking at this we can see that there is a wide range for what different methods consider to be the “best” number of clusters depending on the various criteria each uses. We’ll come back to this again later.

The first columns of the diabetes data set are the predictor, or X, variables and the last column is the class variable, i.e. the response or Y variable. The following commands set up objects for these variables. The object predictorX should be a 768 x 8 array with 8 columns, one for each of the predictor variables. The object predictorY should be a 768 x 1 array. One thing to always remember in R is to use the set.seed() command so that your results are reproducible.

* set.seed(6395)
* responseY <- diabetes[,9]
* predictorX <- df[,1:8]

The R command kmeans() partitions points into k groups so that the sum of squares from each point to the cluster center is minimized. Kmeans() uses the Hartigan and Wong (1979) algorithm by default. So, we could proceed with the cluster analysis from here. However, one of the things we have discussed repeatedly is the challenge in visualizing anything beyond two-or three-dimensions. For our diabetes data we have an 8-dimensional data set. This is not something we can plot on a piece of paper easily.

One thing we can do to make the visualization of the clusters easier is to find the principle components and conduct kmeans() using these principle components rather than over the entire eight dimensional data object predictorX.

So let’s begin by finding the principle components using the princomp() command. The princomp() command is a bit different than the prcomp() command that you learned from Chapter 14 of Peng’s EDA book. It is somewhat less stable but provides much more information. Also, you’ll want the princomp() function to work on the correlation table; so, set cor=T.

* pca <- princomp(predictorX, cor=T)
* pca$sdev

There are several things we want to know from our principle component analysis. Perhaps the most important thing to know is how much of our original data is accounted for in the principle components. “pca$sdev” provides the standard deviations of the principal components in order from the greatest standard deviation to the least standard deviation. In theory, to get the variance, or the amount of the original data accounted for by each principle component we need only to square the standard deviations. But, this can actually be misleading using the information generated by the princomp() or even the prcomp() commands in R. It is better to use the summary() command to look at the overall variance, or how much each principle component explains of the variance in the data. Following the commands from before, we can do this and plot the information as follows:

> summary(pca)

Now, we need to and can decide how many principle components to use in our analysis. If we just use the first two principle components then:

* pc.comp <- pca$scores
* pc.comp1 <- pc.comp[,1]
* pc.comp2 <- pc.comp[,2]
* x <- cbind(pc.comp1, pc.comp2)

The last command column binds the first and second principle components into the matrix x. If you don’t understand how R column binds and row binds you will want to review that.

Now we’ve got a number of ways to consider the number of clusters to use and the principle components, so, we can proceed with kmeans() to plot the clusters along with their centers. There are actually a number of ways to consider how many clusters to use beyond just what pure mathematical analysis yields, although in a sense it should turn out to be the same. For example, the class variable that tells whether or not a participant has diabetes is either 0, no, or 1, yes. In a way that suggests that there are only 2 clusters. However, there are eight predictor variables. So, we can imagine that these predictors may cluster in different ways to predict whether or not someone has diabetes. So from an exploratory data analysis point of view, let’s just look at the solution using a variety of numbers of clusters. Let’s start with two clusters as follows:

* set.seed(1234)
* cl <- kmeans(x, 2)
* plot(pc.comp1, pc.comp2, col=cl$cluster)
* points(cl$centers, pch=16, col="green")

Which shows a plot with two clusters and their respective centers designated with green dots. Now, let’s look at using three clusters as follows:

* set.seed(2345)
* cl <- kmeans(x, 3)
* plot(pc.comp1, pc.comp2, col=cl$cluster)
* points(cl$centers, pch=16, col="blue")

And again with eight clusters as follows:

* set.seed(3456)
* cl <- kmeans(x, 8)
* plot(pc.comp1, pc.comp2, col=cl$cluster)
* points(cl$centers, pch=16, col="cyan")

Last, we’ve set the number of clusters k = 13.

* set.seed(4567)
* cl <- kmeans(x, 13)
* plot(pc.comp1, pc.comp2, col=cl$cluster)
* points(cl$centers, pch=16)

You can list the actual location of the cluster centers using the command:

> cl$centers

For example, the output with 13 clusters might look like:

|  |  |
| --- | --- |
| pc.comp1 | pc.comp2 |
| 1 -3.7845829 | -1.82997218 |

2 -0.5919884 0.04779155

1. 0.4213073 1.71606841
2. 1.2519539 0.67032862
3. -3.1882220 0.67756772
4. -1.4084729 -1.39380205
5. 0.5561967 -0.84983737
6. 2.9454190 0.06105810
7. -1.3865936 1.12360068
8. 1.4928372 -0.69496693
9. -2.4498710 -0.30526698
10. -0.6434765 2.62740683
11. -0.2027170 -1.41481864

These are the locations that are plotted as the points on plots as we’ve gone through this exercise. If you’ve assigned the results of each of these calculations to a different object you can easily make comparisons between them. You can also look at the number of points of the class variable in a table. Again, you can assign the output of these tables to different objects for comparison.

> table(diabetes$Class, cl$cluster)

Where the output for 13 clusters might look like:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **11** | **12** | **13** |
| **0** | 48 | 44 | 43 | 29 | 48 | 27 | 29 | 16 | 8 | 100 | 5 | 2 | 101 |
| **1** | 26 | 40 | 29 | 5 | 19 | 19 | 34 | 22 | 27 | 11 | 7 | 20 | 9 |

You will want to think about what the values in this table represents (Hint: think frequencies)

The point is that at the point of exploratory data analysis cluster analysis is part math, part coding and part discipline knowledge. To go further we need to apply different levels of mathematical comparison between results, each of which has different assumptions. If we don’t know anything else about the discipline we can only say that the kmeans() clustering looks…, well that’s another question isn’t it.