L9: Machine learning exercises

Course: *Pharmaceutical bioinformatics with sequence analysis*
Lecturer: Phil Harrison
Department of Pharmaceutical Biosciences
Uppsala University
Spring 2019

Rstudio and R scripts

If you don't already use it I would highly recommend using `RStudio` (https://www.rstudio.com) and write two scripts for the two exerises below, e.g. "supervised_ml_exericse.r" and "unsupervised_ml_exercise.r". I planned to give a little demo of RStudio at the end of the lecture - if I forgot to then remind me now :)

A good practice, or at least something that I do, is to write the following as the first line in an R scripts to clear the working directory:

```
rm(list=ls())
```

Supervised machine learning exercise

Two major machine learning strategies are supervised learning and unsupervised learning. In supervised learning the machine is given both input and output (or target) data. In pharmaceutical bioinformatics, typical input data may be numerical descriptions of chemicals or structural features of a series of molecules. Output data can be some biological properties of the molecules, determined by their structure. For instance, we may study how structural changes influence ADMET properties (absorption, distribution, metabolism, excretion, and toxicity) or the interaction affinity between the molecule and some receptor protein.

In this exercise we will apply *partial least-squares (PLS)*, *random forest (RF)*, and *support vector machines* (SVMs) for regression to predict aqueous solubility of a series of molecules. The following R code will install the packages we will require for this exercise (note: you only have to do this once to install the packages, so you can skip any for those you may already have):

```
install.packages("rcdk") # Computing molecular descriptors
install.packages("caret") # Spliting into training and test sets
install.packages("pls") # PLS regression
install.packages("randomForest") # Random Forest
install.packages("e1071") # Support Vector Machines
```

Dataset

We will explore a dataset of aqueous solubility of 1142 diverse chemical compounds, which is available at the web address http://pele.farmbio.uu.se/practicalpharmbio/resources/solubility.csv. You may look at the referenced publication http://pubs.acs.org/doi/abs/10.1021/ci034243x to find out more about the original data (it's published as Supporting Info) and the conducted study.

Let's download the solubility.csv file, and read the data into R.

```
web_ad<-"http://pele.farmbio.uu.se/practicalpharmbio/resources/solubility.csv"
download.file(web_ad, destfile = "solubility.csv")
solu <- read.csv("solubility.csv", stringsAsFactors=F)
```

The easiest way to learn what's in the dataset is to use `str(solu)` and `summary(solu)`:

```
str(solu)
summary(solu)
```

This is a data frame containing measured and predicted solubilities for a set of 1142 chemical compounds. Column one, Compound.ID, contains the names of the compounds, and column four holds the SMILES string for each compound.

The function `names()` gets the column names, and prints them on the screen:

```
names(solu)
```

This function can be also used for changing the column names, and to save some typing we will rename the second and third columns in our dataset to "measured" and "predicted", respectively:

```
names(solu)[2:3] <- c("measured","predicted")
```

What we basically have is a measured property of 1142 compounds together with the predictions from a model built by the authors of the referenced paper. Out of curiosity we can check how well the predictions correlate with the measured values by plotting them and computing the correlation coefficient:

```
plot(solu$predicted ~ solu$measured)
cor(solu$predicted, solu$measured)
```

We see a relatively good linear relationship with a corrrelation coefficient of about 0.91. We can also plot a histogram of the measured solubility to see the variable distribution:

```
hist(solu$measured)
```

As expected, the values are fairly normally distributed around the mean of -3.05 (the exact mean value can be seen in the output of `summary()` above), but there are some compounds with very low solubilities.

Computing chemical descriptors

With this information in hand we can ask ourselves: can we build a better model for the aqueous solubility of these compounds? To build a model we need a response variable and some features, or predictors, which describe the chemical structure of our compounds. Using the SMILES of the compounds we can generate molecules and compute descriptors. To save some computational time we can take a random sample from the data, let's say, 300 compounds:

```
set.seed(123)
solu.300 <- solu[sample(nrow(solu), size=300),]
hist(solu.300$measured)
```

A histogram of "measured" shows that our sample is a good representation of the original set - we have similar distribution and the it includes the compound(s) with solubility close to -12.

Let's now load `rcdk`, generate molecules from the SMILES column of `solu.300`, and select some descriptors to calculate.

```
library(rcdk)
solu.300.mols <- parse.smiles(solu.300$SMILES)
allDescr <- unique(unlist(sapply(get.desc.categories(),
get.desc.names)))
allDescr
```

Of the 50 possible descriptors, we will calculate the following:

XLogPDescriptor
WeightDescriptor
AromaticBondsCountDescriptor
TPSADescriptor
CarbonTypesDescriptor
HBondDonorCountDescriptor
HBondAcceptorCountDescriptor

```
solu.300.descr <- eval.desc(solu.300.mols,
allDescr[c(3,4,14,25,39,43,44)])
```

Note that the list of descriptors `allDescr` on your computer may give different ordering, therefore you need to replace the numbers in `c(3,4,14,25,39,43,44)` accordingly.

For a first look at the resulting dataset we can use `str()`:

```
str(solu.300.descr)
```

Our compounds are thus described by 15 features (descriptors) and most of them are discrete variables counting things like number of hydrogen-bond forming groups, carbon atom type, number of aromatic bonds.

Finally, we can add response data.

```
solu.300.data <- cbind(solu.300$measured, solu.300.descr)
names(solu.300.data)[1] <- "measured"
str(solu.300.data)
```

Splitting the data into a training and testing set

We will apply a strategy which allows for easy model quality assessment. The idea is quite simple: we divide our data into a training set and a validation set, use the training set to build a model, and make predictions on the validation set. This is called an external validation, because we assess the model performance using external data, which was not used in building the model. The proportion of data that should be used as a training set depends mainly on the initial data size, and is quite subjective. Also, the exact procedure of splitting the data can differ, and I have chosen to introduce you to a method using the R package `caret`. The following R code installs and loads `caret`, uses

the function `createDataPartition()` to pick 75% of the observations in the form of indexes stored in `inTrain`, and splits the original dataset into **training** and **testing** sets.

```
library(caret)
inTrain<-createDataPartition(y=solu.300.data$measured,p=0.75,list=F)
training<-solu.300.data[inTrain,]
testing<-solu.300.data[-inTrain,]
dim(training)
dim(testing)
```

These 227 compounds in **training** will be used to building our model, and the 73 in **testing** for external validation.

Note: `createDataPartition()` might give you slightly different partitioning on different operating systems. You will also get different results if you set seed to other value than in this script.

Partial Least Squares (PLS) for regression

For fitting the PLS model, we will use the function `plsr()` from the R package called `pls`:

```
library(pls)
```

To fit the model we can use the following expression:

```
solu.pls<-plsr(measured ~.,ncomp=10,data=training,validation="LOO",
scale=T)
```

In this expression we specify that "measured" is modeled as a function of all the other variables in `training`. ncomp=10 sets the number of components that should be used in building the model, and with validation="LOO" we specify that leave-one-out (LOO) cross-validation should be performed. Leave-one-out cross-validation means that the algorithm will build 227 models (this is the number of observations in the training set), each time using 227-1 observations for fitting the model and the remaining 1 observation for testing the model.

We can print the model details and validation results using `summary()`:

```
summary(solu.pls)
```

The RMSEP part, or root mean squared error of prediction, gives the results from the cross-validation precedure. We see that the model with only the first component has RMSE around 1.15, and including more components reduces the RMSE of the model. Plotting the RMSEPs against the number of components gives a good visual feeling for the number of significant components:

```
plot(RMSEP(solu.pls), legendpos = "topright")
```

We see that 4-6 components are probably enough for this model. Now if we pass the model object `solu.pls` and 5 components to the function `plot()` it will plot the cross-validated predictions with 5 components versus the real values of the response:

```
plot(solu.pls,ncomp=5, line=T)
```

Predicting on the test set

So far we have only looked at the validation results from the training data. Using the test set of compounds will give us a good assessment of how our model performs on new data:

```
pred.pls <- predict(solu.pls, ncomp = 5,newdata = testing)
pred.pls
```

We can plot the predicted values against the real values to check how well they correlate:

```
plot(testing$measured, pred.pls)
abline(c(0,1), col="red")
```

The correlation coefficient between predicted and real values can be calculated as:

```
cor(testing$measured, pred.pls)
```

Variable importance

The importance of the variables in PLS can be seen by inspecting their loadings on the components. These loadings are similar to those used in Principle Components Analysis (PCA). More details on PCA and loadings will appear later in the *unsupervised machine learning exercise*. Our variables loaings can be extracted from **solu.pls** using the function `loading.weights()`:

```
solu.pls.load <- loading.weights(solu.pls)
dim(solu.pls.load)
```

Since we have 10 components and 15 variables, all the loadings are returned as a 15 x 10 matrix. The first column is of course the first component and easily visualize the loadings in a barplot if we sort them and use the function `barplot()`:

```
barplot(-sort(solu.pls.load[,1]))
```

The barplot suggests that XLogP has the higher impact on the first component of the model compared with the rest of the predictors/descriptors.

Random Forest (RF) for regression

To use Random Forests in R you need to install the package `randomForest` and load it:

```
library(randomForest)
```

For fitting the model we use the function `randomForest()` in the usual R way, supplying a formula and a dataset. We want to model the column named "measured" against the rest of the variables in the training data frame, and we save the result in a new object (of class randomForest) `solu.rf`:

```
solu.rf <-randomForest(measured ~ ., data=training, importance=TRUE)
```

The argument importance = TRUE tells the function to assess the importance of the predictors. Printing `solu.rf` shows us some details about the model, the mean of squared residuals and the percentage of variance explained:

```
solu.rf
```

The algorithm has grown 500 trees, and the number of predictors used in each bagging split is 5, which is 1/3rd of the total number of predictors. The default number of randomly sampled variables in Random Forests regression is **p/3**, where **p** is the total number of variables. % Var explained shows us that the model explains around 84% of the total variation.

To see how well the model performs on the test data, we can use function `predict()`. When we plot the predicted versus real values, we get this nice linear relationship with correlation coefficient close to 0.94:

```
pred.rf <- predict(solu.rf, newdata = testing)
plot(testing$measured, pred.rf)
abline(c(0,1), col="red")
cor(testing$measured, pred.rf)
```

Random Forests provides two measures for variable importance, which can be used for interpretation of the model. These measures are calculated during the model fitting, and stored in the "importance" component of the resulting object. We can access their values by typing:

```
solu.rf$importance
```

For visualization we can use `varImpPlot()` function:

```
varImpPlot(solu.rf)
```

It turns out that XLogP has a much greater impact on the model's mean squared error (MSE) than the rest of the descriptors. This finding is a good confirmation of loadings barplot for the first component of our PLS model.

Support Vector Machines (SVMs) for regression

Support Vector Machines are a supervised learning technique for regression and classification that project the input data into high-dimensional space and try to find hyperplanes that separate the data in an optimal way. SVMs are sensitive to the parameter choices (as opposed to random forests), namely gamma, the width of the kernel function (this is the function used for mapping the data into high-dimensional space), and cost, the error penalty parameter.

A popular implementation of SVM is LIBSVM by Chang & Lin
http://www.csie.ntu.edu.tw/~cjlin/libsvm/
and the R package that provides an interface to LIBSVM is `e1071`. Let's load the package:

```
library(e1071)
```

The function to train a SVM model provided by `e1071` package is `svm()` but since we don't know *a priori* the best parameters for the algorithm, we will use the function `best.svm()`.
With `best.svm()` we can make a grid search and find the best prameters for our data.

```
solu.svm <- best.svm(x=training[,-1], y=training$measured,
gamma=c(0.001,0.01,0.1), cost=c(4,8,16,32,64), probability=T,
tunecontrol=tune.control(cross=10))
```

In the function we specify the response variable `y=training$measured` and the predictors matrix `x=training[,-1]`.

Note: in the arguments of `best.svm()` we pass vectors with several values for the two tunable SVM parameters, gamma and cost. Using these vectors the function will do a grid search, i.e., it will build models with different combinations of these parameters, and will return the best performing model. Furthermore, `tunecontrol=tune.control(cross=10)` will use cross-validation with ten partitions. `probability=T` specifies that the model should allow for probability predictions.

When we look at the result we see that the best model has gamma and cost selected from the grid search:

```
solu.svm
```

To check how the model performs on the test set data, we make a prediction and plot the predicted values versus the real values:

```
pred.svm <- predict(solu.svm, newdata = testing[,-1])
plot(testing$measured,pred.svm)
abline(c(0,1), col="red")
```

In the same way as for PLS and Random Forest models, we can calculate correlation coefficient between predicted and real values:

```
cor(testing$measured, pred.svm)
```

Additional tasks

The tasks below are optional. Depending on how much time you have left, you may wish to first go through the *Unsupervised machine learning exercise* below and then return to these.

Task 1

Compare the performance of the three models you have created. Are any of them superior to the others?

Task 2

Create models for the whole dataset of 1142 compounds (for this purpose you only have to modify one row of the code). Does the quality of the models improve along with the size of the data set?

Task 3

Create models adding some more descrciptors, for instance, BCUTDescriptor (which encodes atomic charge, atomic polarizability, and atomic hydrogen bonding ability). Does the quality of the models improve along with the number of descriptors?

Task 4

You may have answered question (1) above simply by visually comparing the graphs and considering the correlation coefficients. An additional and better way to assess this is through the mean squared error. Try to write your own code to compute this for each of the models.

Hint 1: See slide 10 from the lecture.
Hint 2: All you need are the predictions (e.g. `pred.svm`) for each model, the true values (`testing$measured`) and the number of test observations (the length of these vectors).

Unsupervised machine learning exercise

In supervised learning the machine is given both input and output data (as you learnt above). In unsupervised learning the machine is only given the input data. Unsupervised learning aims to build representations of the data, making it simpler to comprehend, to identify patterns and to detect outliers. In pharmaceutical bioinformatics typical input data are numerical descriptions of chemicals or structural features for a series of molecules.

In this exercise we will explore three unsupervised machine learning methods: *Principal Components Analysis (PCA)*, *hierarchical clustering* and *K-means clustering*. The following R code will install the package required for this exercise:

```
install.packages("cluster")
```

Dataset

Download the file `solu.descr.Rdata` from http://pele.farmbio.uu.se/practicalpharmbio/resources/solu.descr.Rdata and load it in R. It contains a dataset named `solu.descr`:

```
web_ad <- "http://pele.farmbio.uu.se/practicalpharmbio/resources/solu.descr.Rdata"
download.file(web_ad, destfile="solu.descr.Rdata", mode="wb")
load("solu.descr.Rdata")
ls()
```

Let us look at the structure and the ten first lines of this dataset:

```
str(solu.descr)
head(solu.descr, 10)
```

The dataset comprises a mathematical description of 150 chemical compounds by 8 molecular descriptors. These descriptors define an 8-dimensional space and each compound is a point in this space. Since we cannot make a plot in 8 dimensions, our dataset is hard to visualize, compared to a

dataset of, for example, two descriptors. Also, we don't really know how much useful information each of the descriptors contain, and whether there is high correlation between some of them. One way to get some impression of this is to look at the correlation between each pair of descriptors:

```
pairs(solu.descr,gap = 0.3, cex = 0.5)
```

This scatterplot matrix gives us such a picture. We see a strong correlation between molecular weight (MW) and molar refractivity (AMR), somewhat weaker positive correlation between topological polar surface area (TopoPSA), number of H-bond donors (nHBDon), and number of H-bond acceptors (nHBAcc), and a weak correlation between TopoPSA, MW and AMR. Negative correlation can be seen between TopoPSA and XLogP, and since ALogp2 is AlogP squared, we see a definite quadratic relationship between them.

The function `cor()` will calculate and show us the numerical values of the correlations, and the numbers confirm what we just observed from the scatterplots:

```
round(cor(solu.descr),2)
```

Principal Components Analysis (PCA)

As you already know from the lecture, Principle Component Analysis (PCA) is a statistical technique used for unsupervised learning. PCA transforms a set of (probably) correlated variables into a set of orthogonal (i.e., uncorrelated) variables called principle components. The transformation is such that the first few principle components capture almost all of the variability of the data, i.e. all the important information. Effectively we end up with a much smaller set of variables that still contain most of the information from the original set.

Let's now apply PCA on our dataset using the `prcomp()` function in R. Note the argument `scale. = T`, which tells the function to scale the variables before the transformation, so that they have unit variance and zero mean. While in general there is no consensus on how to best scale the data and whether it should always be scaled, it suffices for this course to say that PCA is sensitive to data scaling.

```
pca <- prcomp(solu.descr, scale. = T)
names(pca)
```

So, we save the PCA result in a new object, `pca`, and if we type `names(pca)`, we will get the names of the five elements of the PCA result. We can access these elements by referencing their names in the usual R manner. Typing `pca$x` will print a 150 x 8 matrix of the principle components, which is essentially the transformed dataset, our 150 compounds in the space of the principle components (PCs):

```
pca$x
```

The difference between these PCs and the original variables becomes obvious when we apply `summary()` on `pca`:

```
summary(pca)
```

which yields a summary table for the PCs, showing their standard deviations and the proportion of variance they explain. We see that PC1 has the highest standard deviation and explains more than

42% of the data variance, PC2 has a lower standard deviation and captures a bit less of the variance, and so on. The last row of the table represents the cumulative variance, and we can see that the first two components have cumulative variance of 78.6 %. If we use plot function directly on the PCA result, it will produce a barplot of the variance of each component, which gives good visual idea for the dominance of the first two principal components:

```
plot(pca)
```

If we make a plot of our data using the first two PCs we can visualize 78.6 % of the variability of the initial 8-dimensional space in only two dimensions:

```
plot(pca$x[,1:2])
```

Variable loadings

The other very interesting element of the object `pca` (the result of PCA) is "rotation". It contains the loadings of the original variables on the principle components:

```
pca$rotation
```

To understand what they mean, we need to recall how the principle components are computed. It starts with the first principle component expressed as a linear combination of the original variables. This linear combination has to have maximum variance, and the variables' coefficients need to be normalized, i.e., the sum of their squares to equal to 1 (otherwise the linear combination will always have a large value if we choose large coefficients). These coefficients are the variables loadings stored as the "rotation" element of `pca`. We can easily check if the normalization condition holds, if take the first principle component loadings, `pca$rotation[,1]`, calculate their squares:

```
pca$rotation[,1]^2
```

and then take the sum:

```
sum(pca$rotation[,1]^2)
```

we get unity. The same is true for each of the principle components.

The absolute values of loadings show the importance of the variables for our principle components, the higher the variable loading, the more important that variable is to the component. If we look again at the loadings for PC1, we see that TopoPSA, nHBDon, nHBAcc, XLogP, AlogP, and ALogp2 have much higher loadings than MW and AMR.

We have already seen that our observations can be plotted in the space of the first two principle components, but to understand what the principle components mean we need to use the loadings. For this purpose we can use the function `biplot()`. It plots the data in the space of the first two principle components, and additionally it plots the loadings of the original variables on the first two components:

```
biplot(pca)
```

The resulting plot confirms the descriptors relationships that we observed from the pairwise plot and from the correlation matrix, i.e., we have three groups of highly correlated descriptors. Furthermore, we see that the first principle component corresponds mainly to the AlogP and XLogP descriptors,

and slightly less to the TopoPSA group of descriptors. So, compounds with high scores on PC1, like 71 and 139, have high values of AlogP and XLogP and low vaues of TopoPSA, nHBDon and nHBAcc. The second principle component is weighted mostly by the MW/AMR group, much less by the TopoPSA group, and only a little by XLogP. Thus, compounds like 79 and 71 are both on the high side of MW, but 79 has very high value of TopoPSA and very low XLogP, and 71 has very low TopoPSA and very high XLogP.

The plot can also be used to identify outliers among the chemical compounds. For instance, we see that compound 5 has very high value of PC2 but compound 46 has very low value of PC1. We can look up descriptor values for these compounds and compare to the mean values of the given dataset:

```
solu.descr[c(5,46),]
```

```
colMeans(solu.descr)
```

Compound 5 has high molecular weight and radius, whereas compound 46 has low values of XLogP and ALogP and it also contains many H-bond donors and acceptors.

Hierarchical clustering

In cluster analysis we try to put objects into groups (clusters) according to some similarity measure. Each cluster consists of objects that are closer, or more similar, to each other than to the objects in the other groups.

One way to do clustering is to arrange clusters into a natural hierarchy. A common approach of this kind is what is called agglomerative, or bottom-up clustering. It starts with all individual objects as separate clusters, finds the two most similar, groups them into a higher-level cluster, and so on until it reaches one big cluster at the top of the hierarchy, containing all the objects and all the clusters. We don't know in advance how many clusters there will be, and we end up with a tree-like hierarchical structure. This method is called hierarchical clustering, and this example demonstrates how to use the function `hclust()` in R for performing hierarchical clustering.

```
hc <- hclust(dist(solu.descr))
```

We pass the distance matrix to the `hclust()` function and we assign the result to a new object `hc`. The resulting `hc` object is basically a description of the tree structure produced by the clustering, and the simplest way to visualize it is just to plot its dendrogram

(http://en.wikipedia.org/wiki/Dendrogram)

```
hc
```

```
plot(hc)
```

On top of this plot we can use another function, `rect.hclust()`, to cut the dendrogram into k clusters and draw rectangles around them:

```
rect.hclust(hc,k=4)
```

Having a matrix of descriptors representing molecular properties, we can plot any two of them against each other using the function `plot()`. In this plot we can easily visualize our clusters and check how much the clustering is related to these two descriptors. To do this, we first extract the cluster memberships with the help of `cutree()` function. Then we use the same plot expression but we add the color argument and set it equal to the cluster membership. For example, let's plot MW versus ALogP:

```
plot(solu.descr$MW,solu.descr$ALogP)
clusters <-cutree(hc, k=5)
plot(solu.descr$MW,solu.descr$ALogP, col = clusters)
```

Now the points in the scatterplot that belong to the same cluster are colored in the same color. We see that for the most part the clusters are separated by molecular weight and not by ALogP.
One should, however, keep in mind that clustering, just like PCA, is influenced by scaling of variables, and the range of values of MW is much larger than the range of values of ALogP. If you repeat clustering on data where each descriptor is scaled to unit variance, the resulting clusters are different.

```
hc <- hclust(dist(scale(solu.descr, scale=TRUE)))
clusters <-cutree(hc, k=5)
plot(solu.descr$MW,solu.descr$ALogP, col = clusters)
```

K-means clustering

In k-means clustering we need to pre-define the number of clusters **K** that we want. The algorithm will randomly choose K initial points, called cluster centers (centroids), and assign each point of our data to its closest centroid. An iterative procedure will recalculate new centroids for these initial clusters, then it will update the assignments, calculate new centroids, and so on until convergence (until reaching some criteria according to which the centroids do not change anymore).
The R function for k-means clustering is `kmeans()`, and it requires at least two arguments, a data matrix and the number of clusters. Let's set the number of clusters to 5. We will see later how this number can be chosen but now we will just arbitrarily pick 5. Since k-means starts with random centroids, we will set the seed in order to get completely reproducible clustering. Here is the R code:

```
set.seed(11)
solu.kmeans <- kmeans(dist(solu.descr), centers = 5)
solu.kmeans
```

The resulting object `solu.kmeans` is a list of several elements. The most interesting of them from our perspective are "cluster" and "withinss". The "cluster" element is a vector containing the cluster assignments for each compound, and if we print it we can see that the first compound is assigned to cluster 2, the second one to cluster 4 and so on:

```
solu.kmeans$cluster
```

One can get a very good overview of the clusters applying the function `table()` on the cluster element:

```
table(solu.kmeans$cluster)
```

(The same information is stored in "size")

```
solu.kmeans$size
```

The "withinss" element (also a vector) contains the sum of squares within each cluster. This is the sum of squared distances between the compounds in the cluster and the cluster center, and it is basically a measure of how good clustering is. The smaller these sums are for each cluster, the better clustering we have. Particularly interesting is the ratio (between_SS / total_SS = 90.3 %), which can be interpreted as the percent of variance explained by the clustering.

We can calculate this value ourselves by dividing the element "betweenss" by "totss":

```
solu.kmeans$betweenss/solu.kmeans$totss
```

which gives us the same value of 90.3%. You can read more about these elements of the clustering result from the documentation of `kmeans()` (to access it, type `?kmeans`)

Visualization of clusters

We can visualize the clusters plotting them in the 2D space of the scaled distance matrix.

First, we need to obtain a 2 dimensional space. We do this by scaling the distance matrix to 2 dimensions, and we save it as a 2 column matrix, `ddim`:

```
ddim <- cmdscale(dist(solu.descr), k = 2)
```

Type `?cmdscale` in R to read more about the Classical Multidimensional Scaling method.
To plot our 5 clusters in 2 dimensions (defined by `ddim`) we will use a function called `clusplot()`. It comes with the R package `cluster`, wich we installed at the start of the exercies. Below we load it and then pass the objects `ddim` and `solu.descr` to `clusplot()`:

```
library(cluster)
clusplot(ddim, solu.kmeans$cluster, color=T, labels=4, lines=0)
```

Choice of k

In this example we've chosen to set the number of clusters to 5, but this choice is quite arbitrary and not necessarily optimal. We will end this exercise with short discussion of this problem, and one example of a method for determining the number of clusters.

We already mentioned the element "withinss", within cluster sum of squares.

Within cluster sum of squares by cluster:

```
kmeans(dist(solu.descr),centers=5)$withinss
```

These five numbers are the sums of squared distances within the clusters. So, the smaller the numbers, the better clustering we have. In the extreme case of assigning each data point to a cluster, the sums will all be zeroes. Decreasing k will increase the sums of squares, and the extreme case of having only one cluster will give the maximum possible sum of squared distances.

Some prior knowledge of the data may help us choose a relevant value of K, but if we don't have knowledge, there are several methods to figure it out.

We'll now learn (or rather you will figure out) how to use one of these, the *Elbow* method, in R. The simple idea behind this method is to perform clustering with different number of clusters, and then plot the totals of the sums of squares against the number of clusters.

Additional Tasks

As above for the *Supervised machine learning exerices* these additional tasks are optional.

Task 1

Try to write the code to perform this *Elbow* method and plot the results.

First create `wss` (either as an empty object with `NULL` or as a vector with 15 elements, you choose). In this you will store the totals (`sum`) of the sums of squares for each of the k-means runs. Next set up a cycle (a `for` loop) doing clustering with 1, 2, 3, ..., 15 clusters, and save the totals for each of them into `wss`.

Finally plot the result and see how the sums of these sums of squares decline with increasing numbers of clusters. You should see that a sensible number of clusters in this case will be 4, 5 or maybe 6, since adding more clusters doesn't seem to reduce the sums of squares dramatically.

Task 2

Try running the model based on your chosen number of clusters multiple times (e.g. change the above `for` loop to keep the same value of k for each iteration). Are the `wss` totals stable when k is fixed?

When this is not the case it is good practice to do multiple runs, e.g. 20 or 50 and take the result with the lowest value. The `kmeans` function can do this for you automatically by setting, e.g. `nstart=20` within the function.

THE END